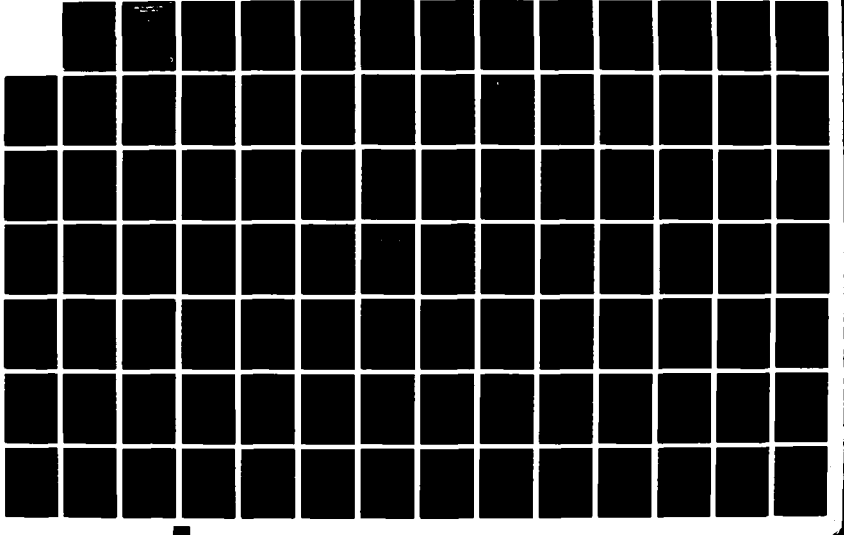
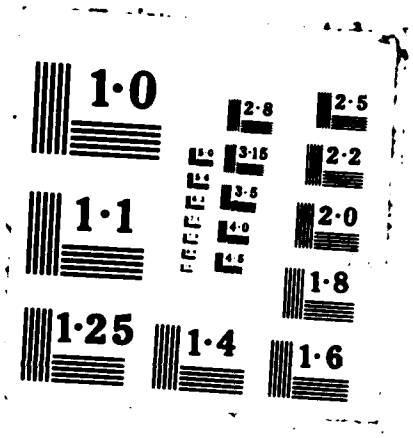
AD-A188 995 DESIGN IMPLEMENTATION AND EVALUATION OF A VIRTUAL SHARED MEMORY SYSTEM IN A MULTI-TRANSPUTER NETWORK(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA S J HART DEC 87 1/2

UNCLASSIFIED F/G 12/5 NL




1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN, IMPLEMENTATION, AND EVALUATION OF A VIRTUAL SHARED MEMORY SYSTEM IN A MULTI-TRANSPUTER NETWORK

by

Simon J. Hart

December 1987

Thesis Advisor: Uno R. Kodres

Approved for public release; distribution is unlimited

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) Code 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey California 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
"DESIGN, IMPLEMENTATION, AND EVALUATION OF A VIRTUAL SHARED MEMORY SYSTEM IN A MULTI-TRANSPUTER NETWORK" (u)

12. PERSONAL AUTHOR(S)
Hart, Simon J.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	1987, December	107

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	OCCAM, Transputer, Multi-Transputer Network, Delay Insertion loop, Virtual Shared Memory.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis presents the design, implementation, and evaluation of a virtual shared memory in a multi-Transputer network. The thesis explores the Transputer Hardware inplementation model and highlights the important details that programmers of such systems may need before being able to optimize such networks.

All the programs and examples presented in this thesis were implemented in the OCCAM programming language, using the Transputer Development System, D700C, Beta 2.0 March 1987 compiler version.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof. Uno R. Kodres	22b. TELEPHONE (Include Area Code) (408) 646-2197 22c. OFFICE SYMBOL Code 52Kr

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
☆ U.S. Government Printing Office 1987-606-24.
UNCLASSIFIED

Approved for public release; distribution is unlimited.

Design, Implementation, and Evaluation of a Virtual Shared Memory System in a Multi-Transputer Network

by

Simon J. Hart
Lieutenant Commander, Royal Australian Navy
B.S., University of New South Wales, 1977

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1987

Author: Simon J Hart
Simon J. Hart

Approved by: Uno R. Kodres
Uno R. Kodres, Thesis Advisor

Richard A. Adams
Richard A. Adams, Second Reader

Vincent Y. Lum
Vincent Y. Lum, Chairman,
Department o Computer Science

James M. Fremgen
James M. Fremgen, Acting Dean
of Information and Policy Sciences

ABSTRACT

This thesis presents the design, implementation, and evaluation of a virtual shared memory in a multi-Transputer network. The thesis explores the Transputer hardware implementation model and highlights the important details that programmers of such systems may need before being able to optimize such networks.

All the programs and examples presented in this thesis were implemented in the OCCAM programming language, using the Transputer Development System, D700C, Beta 2.0 March 1987 compiler version.

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempting to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis are listed below the firm holding the trademark:

INMOS Limited, Bristol, United Kingdom
- Transputer
- OCCAM
- IMS T414
- IMS T800
- Transputer Development System (TDS)

ADA Joint Program Office, United States Government
- ADA

TABLE OF CONTENTS

LIST OF FIGURES

DEDICATION

This thesis is dedicated to:

my wife, Susan,

and

our son, Timothy.

I. INTRODUCTION

A. BACKGROUND

A union of language research by CAR Hoare [Ho79] and the advances of VLSI has produced a unique microprocessor architecture, called the Transputer, which was developed in the United Kingdom by INMOS corporation. The AEGIS Modelling Laboratory at the Naval Postgraduate School has considered the Transputer as a very attractive architecture for future weapons systems control. Weapons systems control computers in general have the following characteristics:

1. They are physically distributed,
2. They require fault tolerance,
3. They are required to be powerful enough to handle very high rate of data now produced by sophisticated sensors,and
4. They are required to be flexible and extendible.

For the above reasons, multicomputer architectures are an attractive option for future weapons systems. Parallel systems intrinsically provide three of the above requirements for a weapon system architecture: high performance, fault tolerance, and extensibility. These features are attained by synchronizing and coordinating the distributed multicomputer network to meet the required aim of the application. Each of the processors must communicate with each other. The most common methods of interprocessor communication are shared memory and message passing. Shared memory communications allows data written to memory by one processor to be read by others. Message passing is mainly point-to-point communication.

There are two major disadvantages of a message-passing scheme compared to a shared memory. The first is that data from a data structure in memory must be organized into messages, which incurs a substantial overhead. The second disadvantage is that there may not always be direct communication links in a network. Messages would have to be passed from one processor to the next until the recipient received it. The disadvantages of shared memory over message passing schemes are three fold. First, in a physically distributed system shared memory would be difficult, if not impossible, to implement. The second disadvantage is that processors must be tied to a high-speed bus which soon becomes a bottleneck as the number of processors is increased. Other critics claim that memory and processor technologies are increasing faster than backplane technologies and that using a bus would prevent the use of the state-of-the-art micro computers and adversely effect any extensibility [Wi87a]. The main objection to the bus structure, however, is that the bus constitutes a single point of failure and therefore is not suitable for fault-tolerant systems.

B. MESSAGE-ORIENTED ARCHITECTURE

The disadvantage of some multi-processor architectures that have been developed to date is that they are a collection of uniprocessors glued by some technique of inter-processor communication mechanism, typically shared memory. These systems are typically single-application oriented and inflexible. A philosophy behind the Transputer is that it is designed to be a multi-processor architecture

with a powerful uniprocessor capability. The inter-processor communications system is actually designed into the processor and not a glue. This message-passing scheme is implemented by four bi-directional links. The bi-directional links are in fact two uni-directional on-chip DMA channels known as link engines. This provides several advantages:

1. An increase in processors increases the number of links which may be operated in parallel,
2. Computation occurs in parallel with message passing. The CPU can operate with minimal degradation during interleaved message passing, and
3. Flexible networks can be designed, since position of processors in a network is not critical.

The links are serially interconnected to reduce the space on-chip and reduce the costs of interconnecting and production. This contrasts to present naval computers in use, such as the AN/UYK 7, where the sophisticated DMA channels are not on-chip and have parallel connections. This increases not only the cost but also the complexity of the system, since there must be additional hardware to integrate the channel to the AN/UYK 7 computers for efficient message passing.

C. HARNESSING NETWORKS OF PROCESSORS

1. Efficiency

The aim of any network is to obtain optimum efficiency and performance. The Transputer and OCCAM permit formidable systems to be built. Given any distributed network, the most difficult problem

programmers face is efficiently synchronizing all processors in the network. In a multitransputer network, processes that communicate with each other do so synchronously. This makes programming simpler but imposes higher system overheads, which may degrade system performance while processors are idle waiting for the next data set to process. Despite optimized inter-process communications hardware, this situation can still exist with with the Transputer. The overheads of the system are not only the idle wait period of network processors but also that of routing messages throughout the network.

2. Programming Languages

To ease the difficulty of programming networks of Transputers, the OCCAM programming language [In87b] has been developed as the high-level language of the Transputer. OCCAM is a structured language which addresses the two main issues, inter-process communications, and parallel processing at the lowest level.

Although several programming languages for parallel processing have been developed, few are commercially available such as ADA and Concurrent Pascal [ShWa87]. Most of the languages use features that are based on the assumption of shared memory, such as monitor and semaphore constructs. ADA for the Transputer is being developed by joint venture involving INMOS and ALSYS software house.

There are now compilers available for the Transputer for other high-level languages such as Pascal, C, and Fortran, but they do not have any ability to exploit parallel activity or communications. Programming productivity could be enhanced by allowing program-

mers to take advantage of these programming languages by using their particular features, such as records and pointers, and "harnessing" them within the OCCAM language features. This is particularly pertinent since OCCAM at present does not have well-developed high-level programming language features.

D. NETWORK TAXONOMY

1. Homogeneous Networks

A homogeneous network is an MIMD network where each node performs the same calculation on separate data. Two examples of harnesses the power of homogeneous networks have been published: the Mandlebrot algorithm [Po85] and the ray-tracing algorithm [At87, Pa87]. These exemplify methods of maintaining full utilization of homogeneous computations throughout a processor network with dynamic load balancing.

2. Non-Homogeneous Networks

A non-homogeneous network is an MIMD network where each node in the network may perform different calculations on separate data sets. Not all applications have the property of all processors using the same algorithm throughout the network. A weapon system is a good example of this. Different processors may have different responsibilities, such as navigation, radar data handling, and displaying the data. Techniques for maximizing throughput in the published examples are not suitable for non-homogeneous systems. Other methods are required. One such method is synchronizing the system using abstract data types called eventcounts and sequencers [ReKa79].

This method of synchronizing distributed systems is ideally suited to a network with low message-passing overheads.

E. OBJECTIVES

In order to explore the methods of programming the Transputer, a full appreciation of its complicated hardware implementation is needed. This thesis intends to build an understanding of the Transputer model through the recently published literature and experimental programming evidence. This should enable readers some insight into what is required to optimize Transputer networks.

Further to the Transputer model exploration, the objective is to create a prototype system to investigate an alternative method of using and synchronizing Transputer networks.

F. THESIS OVERVIEW

The remainder of this thesis is organized in the following fashion. Chapter II describes the two modes of computation of the Transputer, sequential and parallel, a full understanding of which is necessary to understand performance and optimization techniques. Chapter III discusses the communication model of the Transputer in a network. Chapter IV describes the Transputer network architecture and how the network is connected. Chapter V describes the synchronizing mechanism based on eventcounts and sequencers and the underlying details.

Chapter VI discusses evaluation results and summarizes the lessons learned and and issues raised during this exploration. Chapter VII provides conclusions to be drawn and subsequent recommendations.

II. TRANSPUTER HARDWARE IMPLEMENTATION

A. GENERAL

There are three subjects that need to be mastered before programming Transputer networks. These are :

1. The programming language OCCAM;
2. The use of the Transputer Development System for the respective host; and
3. The hardware of the Transputer.

The language OCCAM is straightforward for anyone with a background in structured programming languages. The Transputer Development System, however, is not trivial to master. Until now there has been little detailed information on this aspect due to the rapid development of the system. For the novice, detailed descriptions and examples are contained in [Po87].

The Transputer hardware appears very much straightforward when examining the architectural diagram [In87a, p. 34]. To understand the differences between this architecture and the Intel 80386 or the Motorola 68020 architectures, and to fully harness performance capability, a detailed examination of the Transputer model is necessary. For ease of explanation, the Transputer model has been divided into two naturally distinct models, the sequential and the parallel models.

B. SEQUENTIAL MODEL

The Transputer is a reduced instruction set computer (RISC). The characteristics of a RISC machine are summarized as follows:

1. Operations are always register to register. Only LOAD and STORE instructions access memory only.
2. Operations and addressing modes are reduced. Operations usually occur in one cycle. Addressing modes are relative and indexed (other instructions can be developed from these two basic modes if required).
3. The instruction set is simple and instructions do not cross word boundaries.

/ recent comparison of other RISC machines [GiMi87] showed that the Transputer (T414 20 Mhz) is one of the most powerful RISC architectures available with a 10 MIPS linear performance capability.

1. Register Set

The Transputer CPU is stack based with only six registers: three system registers and three evaluation stack registers. The three evaluation stack registers are labelled A, B. and C.

The system registers are the Workspace Pointer, which indicates the process in execution; the Instruction Pointer, which points to the next instruction to be executed; and the Operand Register, which is used for the formation of instruction operands. This is shown in Figure 2.1.

There are other registers available only to the system to assist in processor management. These are two timing registers and four registers to manage two task queues. These will be discussed in the Parallel Model.

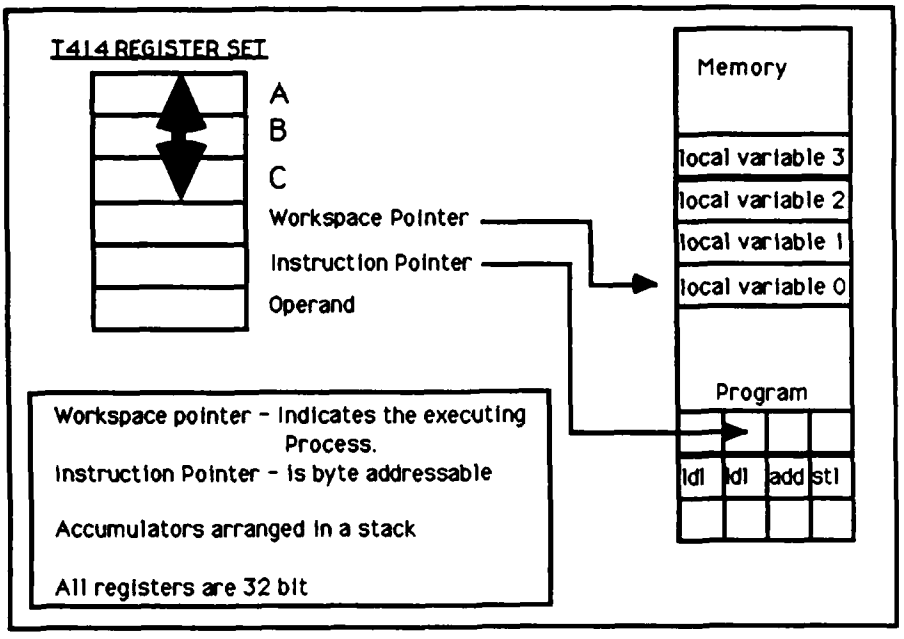


Figure 2.1

T414 Register Set

2. Instruction Format

All instructions are eight bits long and are divided in two. The low-order four bits are the data and the high-order four bits are the opcode or function, as shown in Figure 2.2.

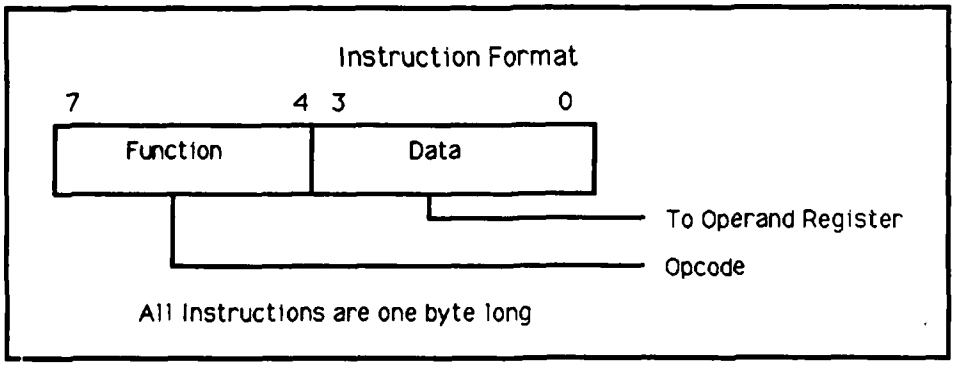


Figure 2.2

Instruction Format

The data is loaded into the lower four bits of the 32-bit operand register and the opcode operates on the entire operand register. This allows 32 bits of data to be used if required, as shown in Figure 2.3.

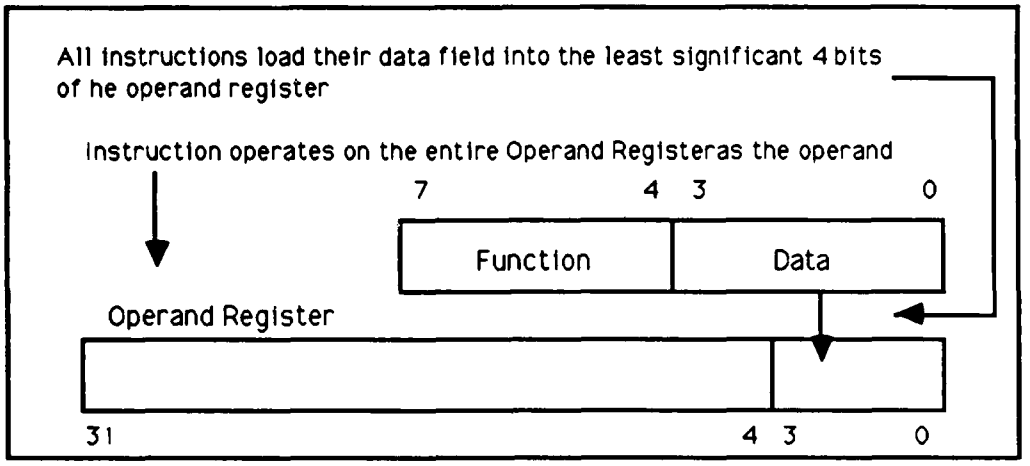


Figure 2.3

Instruction Format

The fact that the function part of the instruction has only four bits allows the Transputer 16 one-cycle instructions. Examination of the instruction set [In87b] will show that 13 of these actually manipulate the processor. These single-byte instructions are the most frequently used instructions, such as store, load, calls, and jumps. The three remaining instructions manipulate the operand register. These are Pfix, Nfix, and Opr. Pfix and Nfix manipulate the operand register. An example of this is shown at Figure 2.3. Opr executes the instruction in the operand.

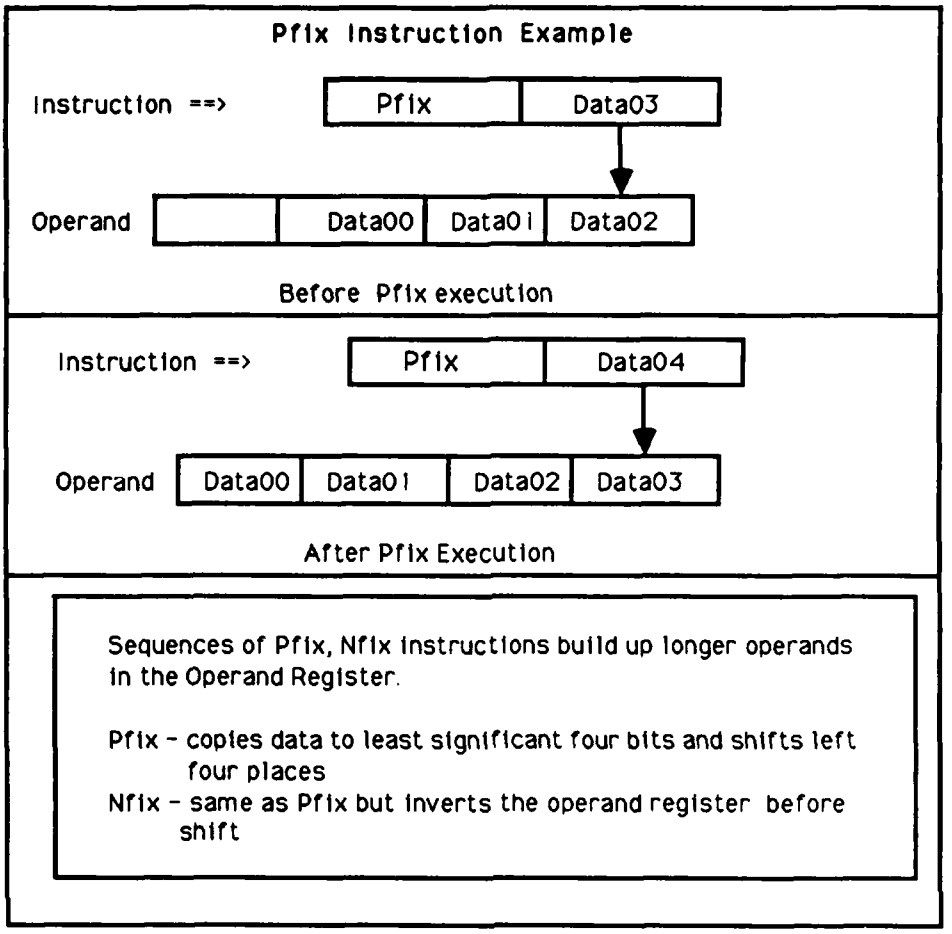


Figure 2.4

Pfix Instruction Example

The simplistic nature of this instruction set facilitates the writing of a disassembler for compiled OCCAM code. There is a disassembler available in the AEGIS modelling group written in PASCAL [Br87]. This has proved most useful to unravel some previous mysteries of the Transputer.

Most arithmetic and logical operations are zero address instructions which operate on the contents of the stack registers. With the ability to manipulate the operand register, there is the

possibility of 2^{32} possible zero address instructions. Further, the Transputer uses a PLA in the decode path which presumably will allow instruction set redesign as the architecture matures [GiMi87].

3. Memory Management

Memory utilization is a feature that the programmer must be aware of to optimize Transputer performance. Memory is divided into on-chip and off-chip memory space. The reason for delineation is that on-chip memory is faster than off-chip memory due to time required for external memory interface. Typical memory cycle intervals for a data fetch are: on-chip memory, two cycles; off-chip memory, a minimum of three but typically four cycles. This means that frequently used data structures should be placed in on-chip memory for maximum performance.

Address space of the Transputer is signed. This is unusual but should improve all logical and arithmetic address operations since there is no need to manipulate the values into one's or two's complement form for each operation.

4. Execution Speed

The Transputer instruction format allows many instructions to be executed in one clock cycle (50 nanoseconds). In reality, about half the instructions require two clock cycles or less. The eight-bit instruction and a four-byte word allow four instructions to be read at one fetch. This is an excellent feature since its provides a virtual four-instruction cache without the cost of on-chip space. Another important advantage of this feature is that it provides an almost total

decoupling of instruction execution speed from memory speed. The only exception to this is when the prefetched word contains an instruction mix of one-cycle instructions. This means that the location of the program is not critical for performance maximization. Details of the implementation of the sequential model is contained in [In87b].

C. PARALLEL MODEL

1. Overview

In either model, the basic execution unit is a process. A process may consist of many sub-processes executing concurrently, time sharing the processor. A process may be allocated one of two priority levels for execution. The higher priority process is uninterruptable. It will run until blocked by communications or timer inputs.

Although not explicitly stated, the parallel model is based on the following assumptions:

a. The shortest context switch is made by saving the least amount of data for any given process,

b. A process must do I/O,

c. A process not doing I/O is in a loop and must eventually execute either a loop end instruction or jump instruction, and

d. A high-priority process needs to execute as soon as it is ready.

Most multi-tasking for any system takes place in an operating system. This is not the case with the Transputer since it is implemented in the hardware. The parallel model requires the following hardware support for implementation :

a. Two timing registers;

b. Four Process Queue Registers; and

c. Special registers for saving some process context switch data.

2. Process Representation

Initiation and termination of processes may be performed either at compile time or dynamically. Each concurrent process is represented by a vector of words in memory called the process workspace. This space is used to hold the local variables and temporary values manipulated by a process. The workspace is organized as a falling stack with end-of-stack addressing. All local variables are addressed as positive offsets from the Workspace Pointer.

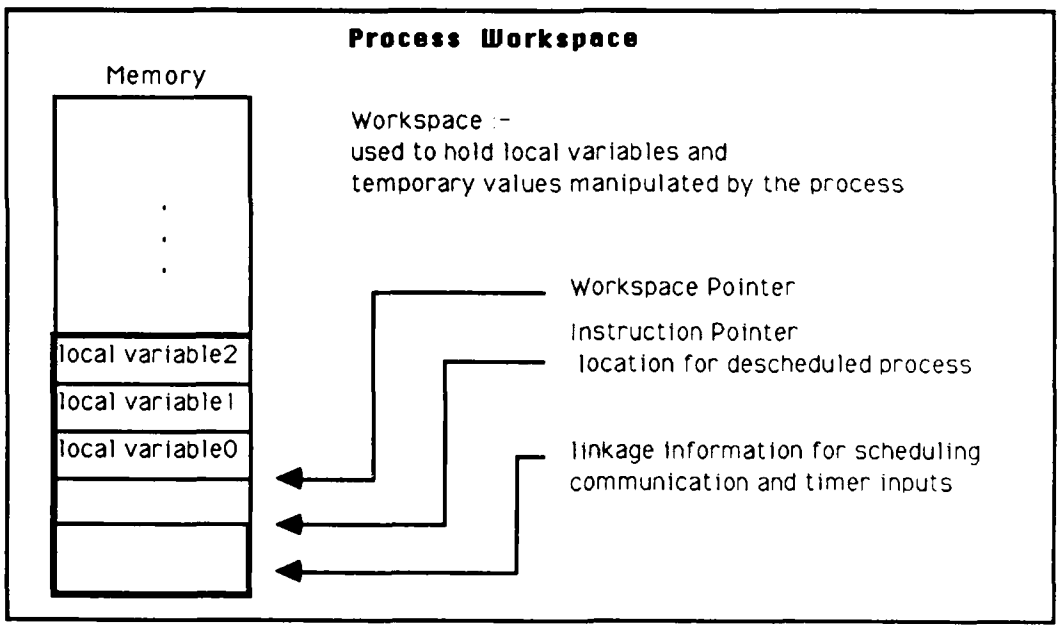


Figure 2.5

Process Workspace

There are other locations associated with the workspace which are used by the operating system. These locations are used for linkage information such as scheduling, communication, and timer inputs and are addressed as negative offsets from the Workspace Pointer. This linkage space varies depending on the the synchronizing constructs used by the process, such as ALT, TIMER, or any communications. When a process is descheduled, the Instruction Pointer is stored in the word below the Workspace Pointer. Details of the linkage area are given in [In87d].

A process is in one of three states: executing, ready, or blocked. The executing process is found by examining the contents of the Workspace Pointer Register. Ready processes are placed in one of two queues. Blocked processes have their workspace pointers stored in appropriate words which are used to relink these processes to the necessary queues when they are rescheduled. The Transputer maintains two ready queues, one for each priority. Each queue is maintained using two registers; one points to the Workspace Pointer of the head of the queue and the second points to the Workspace Pointer of the process at the tail of the queue. Each process has associated with its workspace a word which indicates the next process in the ready queue. A diagram showing the logical structure of this organization is shown at Figure 2.6.

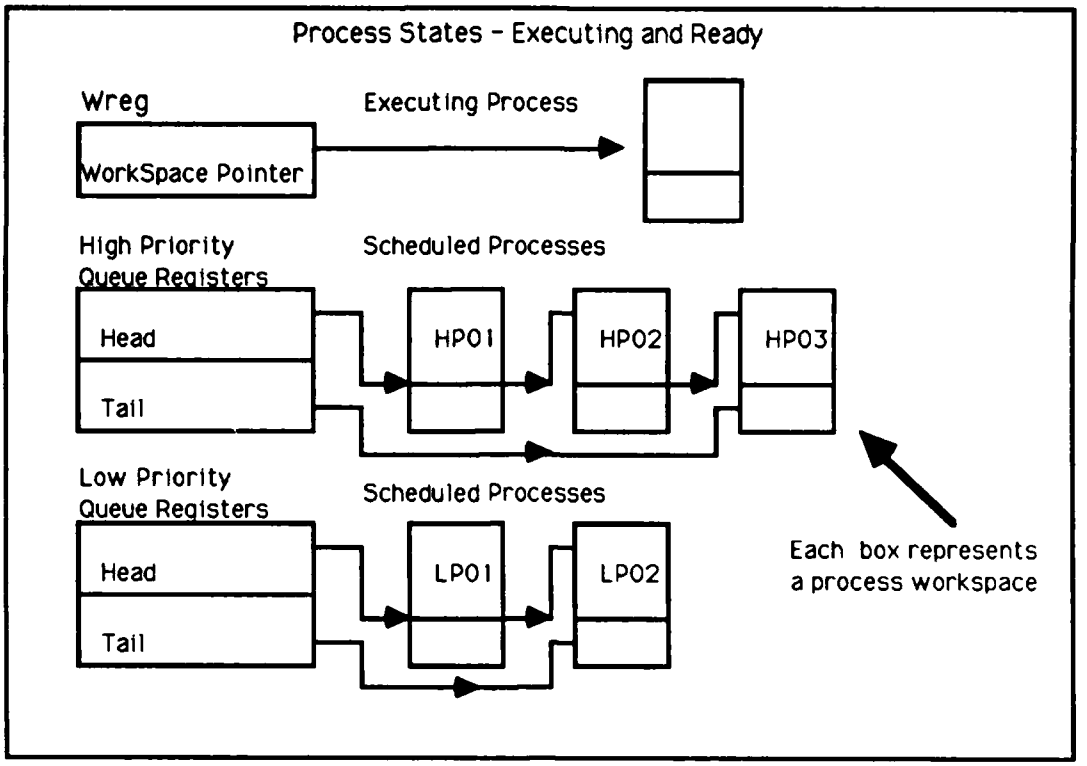


Figure 2.6

Process State Registers

3. Process Priority and Interrupts

When a high-priority task is ready and no other high-priority task is executing, it preempts any low-priority task that may be executing. Generally, this takes place at the end of the current instruction. Some instructions are interruptible; for example, block move or I/O instructions. Full details of interruptible instructions are in [In87d p. 30]. This preemption constitutes an interrupt. The state of the low-priority process is saved in special system memory locations at the low end of on-chip memory and the workspace pointer is placed at the head of the low-priority queue. The process context

switch time is low since it need only save six registers and memory allocation for saving the state is on-chip.

4. Process Scheduling

There seems to be a widespread misunderstanding that the low-priority processes are time-sliced. This is a misnomer since there is no fixed period for process descheduling. The mechanism works according to the following rules:

a. A process will be descheduled when it attempts to synchronize (via communication) with a process that is not yet ready to synchronize, or when it attempts to communicate externally using the hardware links.

b. If a process does not perform any I/O for more than one time-slice period, it will be descheduled at the next descheduling point. Details of these instructions are given in [In87b p. 66].

5. Time Slice Periods

A time-slice period is defined as 1024 ticks of the high-priority clock. When the one-time slice period has occurred, the processor will attempt to deschedule the low-priority process that has been executing. Each time the process reaches a descheduling point, the processor checks to see if a time-slice period has elapsed. If so, the process is descheduled and added to the end of the appropriate list. In short, the minimum period of time for "time-slicing" is one millisecond, with the expected maximum period being two milliseconds.

6. Overheads

There is full instruction level support for context switching which provides very low overheads. Sub-microsecond context switch

times are quoted by INMOS [In87b, In87c] for a 20 Mhz processor. Experimental data has shown that overheads are one microsecond on the average.

7. Programming Practices

It is important to understand the parallel model since it does have an impact on high-level programming practice in allocating priorities to processes to ensure efficient process execution. The lesson is to avoid placing a computation-bound algorithm in a high-priority process. High-priority processes should be kept short and I/O bound; otherwise, network performance will be sub-optimal.

This aspect of the model was investigated in the following manner. A simple calculation process which ran for a known execution time was placed as a background process to a high-priority process. The background process executing time was delayed by the length of the high-priority process, which validated this aspect of the model. Further investigation proved that placing a computation-bound process in the high-priority queue did in fact degrade the performance of other high-priority processes. The conclusion from the investigation showed that high-priority process allocation should be given to message-passing code. This allows all network messages to be passed as quickly as possible. Further discussion and examples are provided in [At87].

D. TIMERS AND TIMINGS

1. Overview

The Transputer has two 32-bit timers. The timers provide accurate process timing and allow the programmer to deschedule processes explicitly until a specified time. Implementation is shown in Figure 2.7.

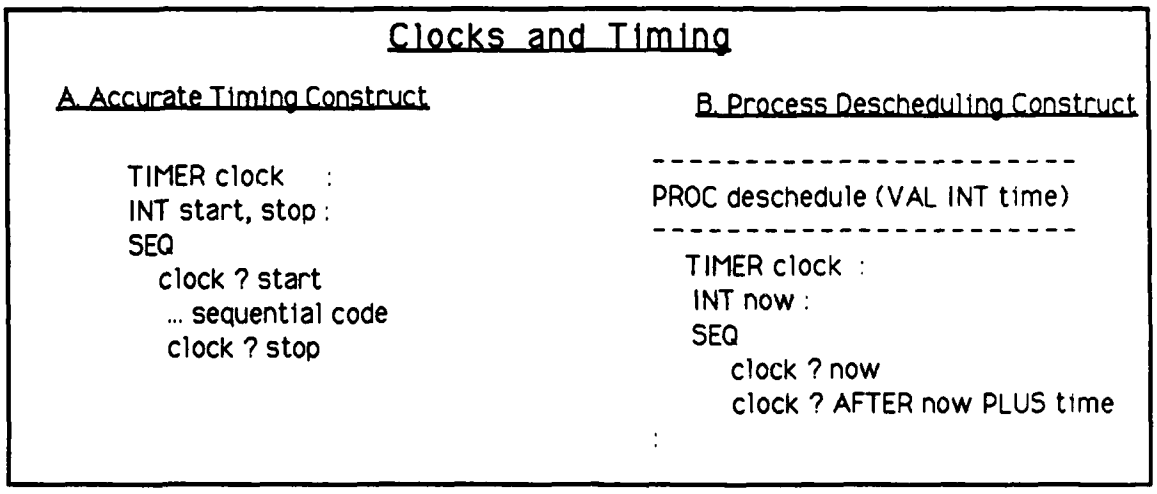


Figure 2.7

Clocks and Timing

A diagram showing processes on the timer queues is shown in Figure 2.8. The main point to note here is that use of timing queues is expensive in cycle time (30 cycles) and is dependent on the length of the queue.

2. Use of Timers for Timing Constructs

Timing constructs should be used very carefully. This is especially the case with parallel constructs and timing communication

rates. Thorough investigation into the use of timers showed the following results:

a. An elapsed time construct as in Figure 2.7 provides elapsed time from start to finish. However when used in a parallel construct, this also includes context switch overheads and time spent in the queue and not just execution time of that process.

b. Enveloping a PAR construct with an elapsed time construct includes spawning, executing, and context switch overheads needed to execute that construct.

c. Timing communications constructs (either input or output) cannot be considered accurate due to the nature of the communication implementation. This is especially so with external link communications, since the link engine is a DMA channel and communication is decoupled from the processor until finished.

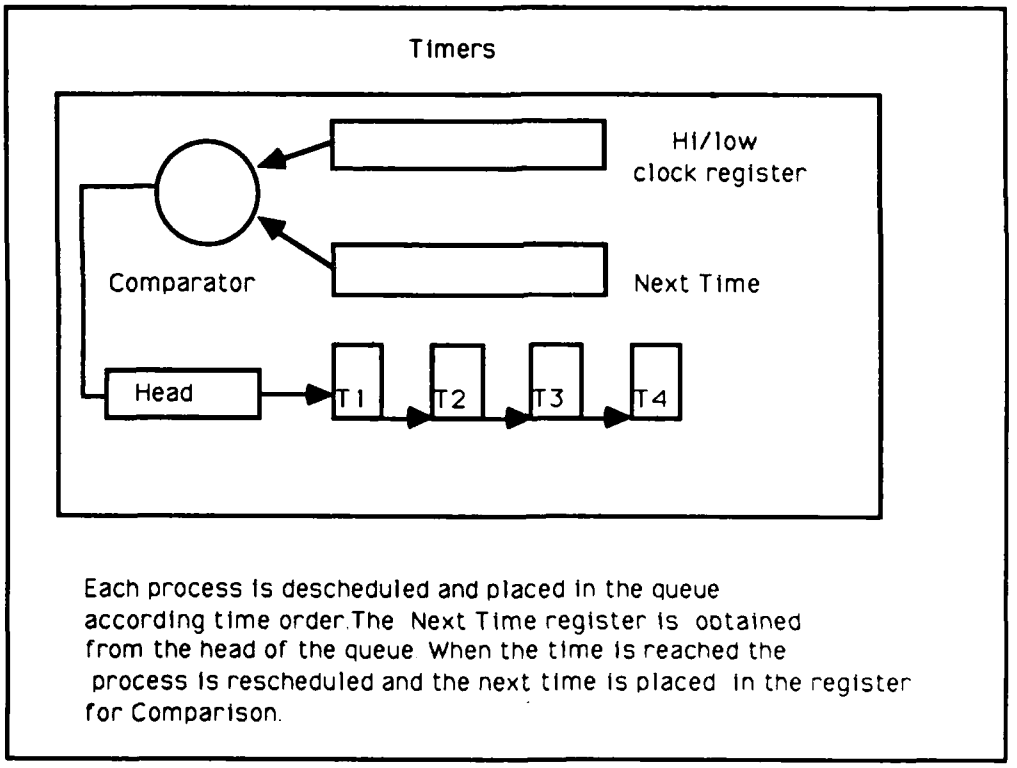


Figure 2.8

Timers

3. Programming Practice

The use of the elapsed time has been useful to accurately time sequential in-line code and useful for estimates of time for parallel construct code. For the most accurate timing of in-line code, it is recommended that the process be run at high priority so that the one microsecond clock is used. An example of this is shown in Figure 2.9.

```
Accurate Timing Code

PRI PAR
  TIMER clock :
  INT start, stop :
  SEQ
     clock ? start
     ... timing code
     clock ? stop
   SKIP
```

Figure 2.9

Accurate Timing Listing

It is also worthy of note that using timer constructs is expensive in cycle time. For example the instruction *timer input* has a worst case of 30 cycles. This may be important when programming real-time programs.

Processes executing in parallel have a requirement to communicate with each other. This aspect of the Transputer is investigated in the next chapter.

III. COMMUNICATIONS

A. GENERAL

The most powerful aspect of the Transputer is that it is specially designed for the two main criteria of multi-processor architectures: parallelism and inter-process communication. Understanding the communications mechanism of a network allows a programmer to use these features to advantage in the pursuit of optimizing network performance. This chapter discusses the essential performance issues of Transputer communication.

B. BASIC NOTIONS

In the Transputer, concurrent processes communicate synchronously by using channels. Communications only occur when both the sending and receiving processes are ready. This model was developed by C. A. R. Hoare in the experimental language CSP [Ho79]. OCCAM contains a construct which implements an abstraction of CSP synchronous communication. This abstraction is called a channel. A channel may be described as an unbuffered, unidirectional connection between two processes. The construct is the same for internal or external inter-process communication. There is a difference, however, in the way each method of communication is conducted. Internal communication is achieved simply by memory-to-memory data transfer. External communication is conducted by one of the 8 DMA link engines. Each link engine corresponds to an external channel. Each Transputer link has two unidirectional channels.

C. INTERNAL COMMUNICATION

A channel is a single word in memory. This channel is assigned to the two communicating processes by the programmer. At compile time this channel is assigned a specific word in memory. This word is used to hold either an address to a process' workspace or the special value 80000000H (the minimum integer) which represents nil. All channels are initialized to nil at compile time.

To exemplify the communication, assume there are two concurrent processes, Alpha and Beta, in a single Transputer. Alpha is the sender and Beta is the receiver. Suppose Alpha is ready to send, and Beta is not yet ready to receive. When Alpha attempts to communicate, three items are loaded into the stack: the address of the dedicated channel, the address of the message data structure, and the length of the message. Once this information is loaded, the output instruction is executed.

Upon execution of this instruction, the channel word is examined. If it contains the nil pointer, it is the first process to attempt to communicate and accordingly places its Workspace Pointer (contents of the WReg) and Alpha's instruction pointer (contents of the IReg) and the message length and pointer in the linkage area below Alpha's workspace. Alpha has now been descheduled and is in a blocked state.

Alpha will remain blocked until such time as Beta attempts to communicate via the dedicated channel. When Beta attempts to receive, the same information is loaded into the evaluation stack and the value stored in the channel word is checked. This channel word

now contains the pointer to Alpha's workspace. The processor now conducts the transfer by block move, after which the channel word is re-initialized and Alpha and Beta are rescheduled. The communication process is symmetrical; if Beta had become ready first then exactly the same procedure would be followed but in the reverse order.

D. EXTERNAL COMMUNICATION

When a process wants to communicate with an external process, it does so by using one of the 8 DMA link engines. The link used is explicitly allocated by the programmer. This selection is dependent on direction of communication and the network topology. The link arrangements are shown in Figure 3.1.

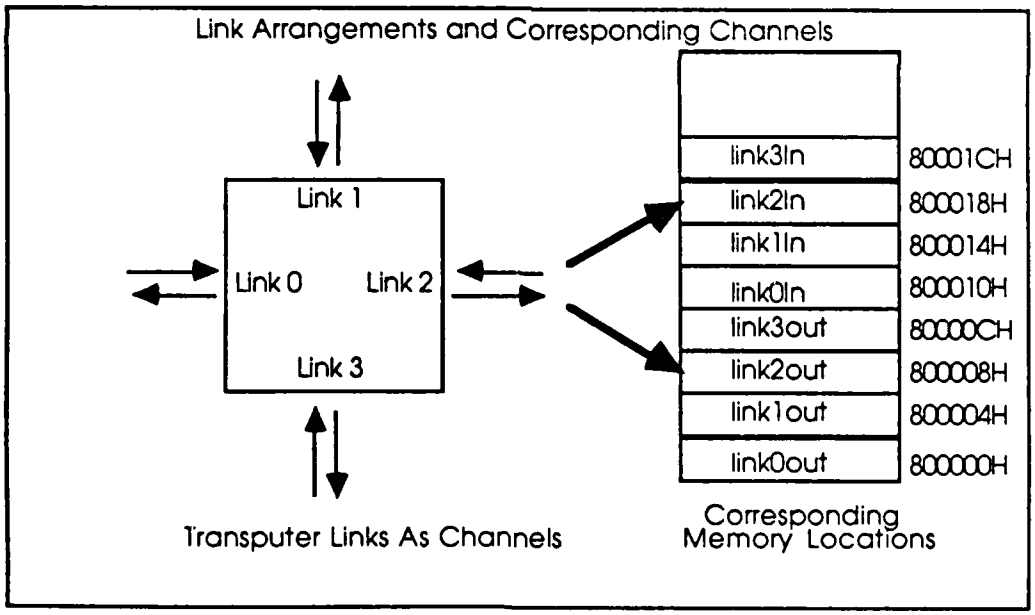


Figure 3.1

Transputer Link Arrangements

Each link has a dedicated channel word which is placed in one of the eight lowest words of memory in the Transputer. Given exactly the same example as above, but each process in separate Transputers, the procedure for communication is followed as described above with the following exception. The use of special channel words is detected and the three pieces of information are sent to an autonomous link engine interface unit. This is shown in Figure 3.2.

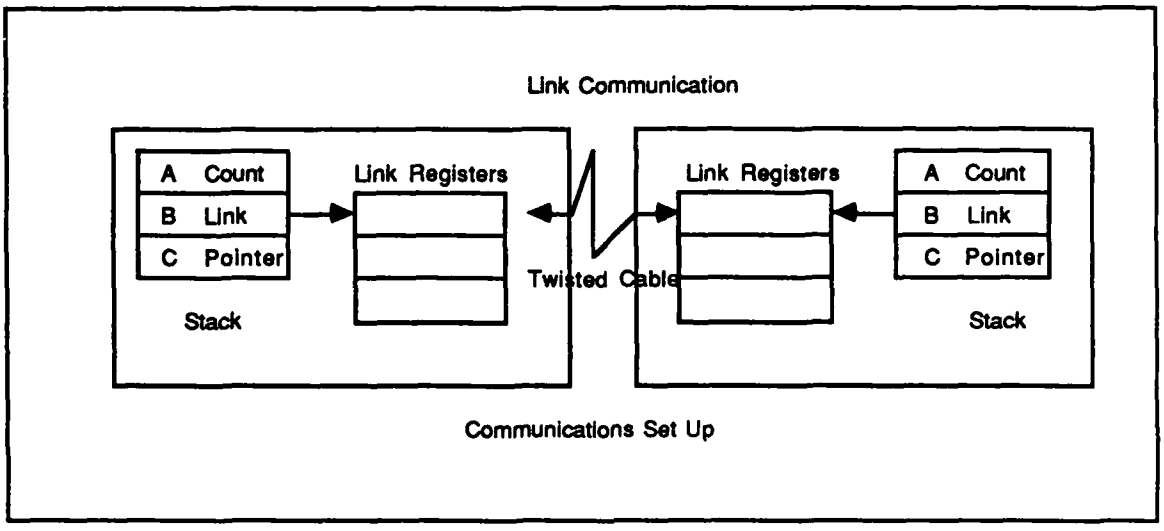


Figure 3.2

Communications Set-Up

Alpha is blocked until the link engine has completed the block transfer. Once the transfer has been completed, both processes are then rescheduled by placing the Workspace Pointers in the link interface units on the appropriate queues, as shown in Figure 3.3.

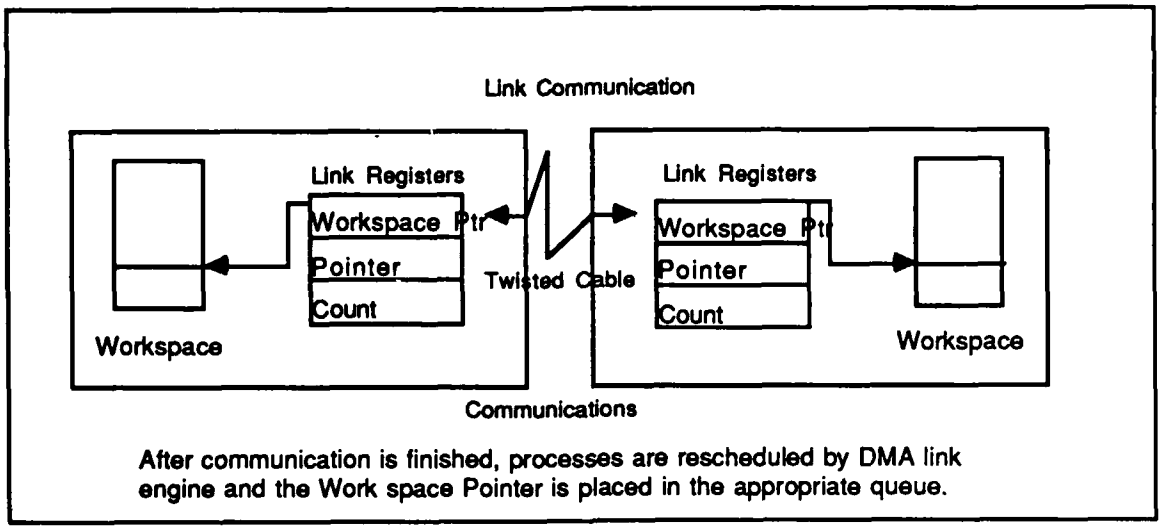


Figure 3.3

Communication Rescheduling

E. LINKS

Access to the links is via the processor controlling the link engine. Each link wire has a separate DMA channel so all engines may be active simultaneously. The DMA engine interleaves all memory requests appropriately. The control registers of the DMA engine are memory mapped. Although the link protocol is an important performance parameter in examining data throughput, this subject is not covered in this paper. [Va87] provides a detailed description of the topic. Other pertinent information is that there is no error checking done with link communications. However, if the length of link twisted cables is greater than 0.9m, suitable error checks should be made.

F. PROCESSOR PERFORMANCE

There are two main areas where link communication will influence processor performance:

1. Communication transfer setup time, which is approximately 21 cycles per message for external links [In87b].
2. DMA link engine cycle stealing, which consumes typically 4 processor cycles every 4 microseconds per link engine.

Significant to processor performance is the link engine's usage of the internal bus during any inter-processor communication and its potential degradation of the processor utilization. Cycle stealing by the DMA link engines yields varying degrees of performance degradation for given instruction mixes. An investigation was made into the use of the internal bus in an attempt to quantify maximum performance degradation for particular instructions.

Discussion with an INMOS consultant [Ma87] revealed that, when the process conducting I/O and the process using the processor are of the same priority, the internal bus gives priority to the link engines over the processor due to their lower bandwidth. A higher-priority process will always preempt lower-priority processes from internal bus usage. This means that, to ensure efficient network communications, processor performance will always be degraded to some degree since computation bound processes should be run at low priority and hence would never have bus priority.

G. EVALUATION

Two operations, *divide* and *block move*, were selected to determine the maximum performance degradation. It was anticipated that

the degradation of the processor would be the greatest for the operation with greatest amount of memory access time. Each background process, consisting of several iterations of instructions, was timed with no link operations and then as a background process with eight link engines in operation. All programs were on-chip. The evaluation program is shown at Appendix A. The results are shown in Table 3.1.

TABLE 3.1

BACKGROUND PROCESS DEGRADATION

TABLE 3.1 - Background Process Degradation

	Block move			Divide		
Iterations	In-line	Background	%	In-line	Background	%
50	4167	4600 (72)	9.0	223	192 (3)	-
100	8333	10688 (167)	22.0	445	448 (7)	1.0
500	41658	44736 (699)	6.9	2217	2368 (37)	6.3
1000	83316	86336 (1349)	3.5	4433	4800 (75)	7.6
5000	416573	420736 (6574)	9.9	22156	25088 (392)	11.0

All results are shown in high level ticks (1 microsecond) for both the In-line and background process execution. They represent the execution time without and with link interference respectively. The bracketed figures are the low level ticks recorded for the background calculation with the four link interference. These low level ticks re converted to high level figures for sake of comparison. The appropriate percentage degradation is also shown. The expected

results were that *block move* instructions would be delayed by 12% and *divide* instructions by 9%. *Divide* operations gave varying performance degradation between 1.0 and 11.0% and *block move* of 3.5 and 22.0%. Within the basic understanding of the Transputer model, this is difficult to explain.

Based on the fact that links transfer one byte of data every 23 bit times and a minimum instruction fetch at 200 microsecond intervals, eight link engines in operation for the T414 Transputer link protocol (16 processor cycles every 4 microseconds), the absolute maximum degradation possible is 25%.

H. PROGRAMMING PRACTICE

Investigation into this aspect of processor performance has shown that in a network, to maximize performance, the largest overhead in message passing is the transfer set-up time. More studies need to be conducted to verify it. Discrete messages should therefore be kept as long as possible, which agrees with [At87].

Further discussion with an INMOS consultant [Ma87] revealed that The Royal Signals Research Establishment, United Kingdom, has studied the overheads of network message passing concerning processor efficiency and the optimum message length was found to be between 10 and 100 bytes.

Understanding the fundamental Transputer models is the first step toward use of the Transputer in a particular system architecture. The next step is harnessing the optimized multiprocessor characteristics. Chapter IV looks at a method to use these features.

IV. MULTI-TRANSPUTER NETWORK WITH GLOBALLY DISTRIBUTED VARIABLES

A. INTRODUCTION

The aim of this chapter is to lead the reader into what motivated the design of multi-Transputer networks with globally distributed variables. We briefly discuss what configuration was selected, and briefly describe the synchronizing mechanism to implement the design.

B. MOTIVATION

One successful method of harnessing multiprocessor systems is implemented using shared memory and global variables. Processes may communicate by means of globally shared variables maintained in a physically shared memory. The reading and writing of these variables is controlled by an operating system which ensures reading and writing is achieved in a carefully synchronized fashion. For example, using the classic producer-consumer paradigm, the operating system will ensure that any writing to a global data structure is completed by a producer before any consumer process can read such a data structure. One such mechanism is described by Reed and Kanodia [ReKa79] and showed its implementation within a shared memory environment. This synchronization is based on eventcounts and sequencers. Such a synchronization system was used in the MCORTEX operating system [Ga86, Ko83], which provided very satisfactory performance results. The features of the synchronization mechanism in particular are well

suited to physically distributed systems such as multi-Transputer networks.

The problem domain is harnessing the network to its full potential, given the two factors of proven multiprocessor synchronization mechanism and a powerful Transputer multiprocessor architecture. Our proposed solution is a network of Transputers which share no physical address space but maintain an equivalent of a physically shared memory system by replicating the global data structures throughout the nodes in the network. Each participating node producing any new value for any global data structure would broadcast this value throughout the network for updating the other node's replication memories. This is called a virtual shared memory system. In this system, network communication is conducted with minimal degradation to each node's processing power. To achieve this, the network must be set up in an optimum configuration to ensure optimal performance.

C. OPTIMAL NETWORK CONFIGURATION

The four links of the Transputer allow flexible network configurations which are application dependent. [Be85] discusses optimal configurations of multi-Transputer networks. The superior network configuration for implementing the virtual shared memory system is the delay insertion loop structure [We80]. The reasons for such a selection are as follows:

1. Addressing schemes overheads are minimal.

2. A transmitting node needs to know the location of any receiving node.

3. Message broadcast is facilitated.

4. Node connections can be established quickly and easily. (This may be software controlled using an INMOS C004 connection scheme [In87e].)

5. A loop configuration allows a high message throughput rate.

6. A loop structure enhances modularity throughout the network.

The major disadvantage of the system is its reliability due to its serial nature. This is recognized but ignored for sake of evaluation. Fault tolerance within this system is another issue. The prototype for the virtual shared memory system therefore uses only a unidirectional ring structure.

Source code for the ring is shown at Appendix B. The ring size is dictated by the structure of the INMOS B003 Evaluation Boards. Consequently, the minimum size is four nodes and increments are in multiples of four. Other network structures are shown and discussed in [Hi].

D. EVENTCOUNTS AND SEQUENCERS

1. Eventcounts

An eventcount [ReKa87] is an abstract data type (ADT) which maintains a count of the number of occurrences of a particular class of events within a system. It is implemented as a non-negative integer variable initialized to zero. Associated with this ADT are three primitive operations as follows:

a. advance (Event.count)

b. read (Event.count)

c. await (Event.count, Threshold.Value)

Advance causes the value of the eventcount to increment by one. This signals another occurrence of an event associated with that eventcount. *Read* returns the present value of the eventcount. *Await* provides a non-busy wait synchronization tool which deschedules a process until such time as the eventcount has reached or exceeded the threshold value. Thereupon it is rescheduled for execution.

2. Sequencers

The sequencer is also an ADT. It is designed to provide total ordering of events within the system which is implemented as a non-negative integer variable. The only operation associated with it is *ticket(This.Sequencer)*. This operation returns the current value of the sequencer and then increments the sequencer value by one. The concept is analogous to the barber shop ticket system when, upon entering the shop, the customer takes a ticket and, when the barber calls his number, he is the next person for a haircut. This mechanism provides mutual exclusion for system resources if required.

E. EFFICIENCY

The software design objective is to minimize processor idle time and maximize system throughput. Multiprocessor systems will suffer reduced efficiency by bottlenecks due to serialized processing caused by inadequate synchronization. With careful attention to the multi-Transputer network architecture and use of eventcounts and sequencers, these bottlenecks may be avoided.

V. DESIGN AND IMPLEMENTATION OF A VIRTUAL SHARED MEMORY IN A MULTI-TRANSPUTER NETWORK

A. OVERVIEW

This chapter attempts to walk through all the design issues involved in designing and implementing a prototype virtual shared memory system in a multi-Transputer network. The aim of the chapter is to document the design decisions so any subsequent work in the area may benefit from both the strengths and weaknesses of these decisions. The design process was an iterative one. Changes were made as an understanding of the models discussed previously became clear. The issues are dealt with from a top-down design view of the problem. Consequently, this chapter is divided logically into system model and assumptions, macro-design decisions, and micro-design decisions.

B. SYSTEM MODEL AND ASSUMPTIONS

1. Node Activities

Each node during system operation will have three major activities:

a. message routing,

b. updating all incoming global data structures, and

c. calculation of data for distribution.

These activities are directly mapped to associated processes labelled filter.data, update, and calculate. Their logical structure is shown in Figure 5.1.

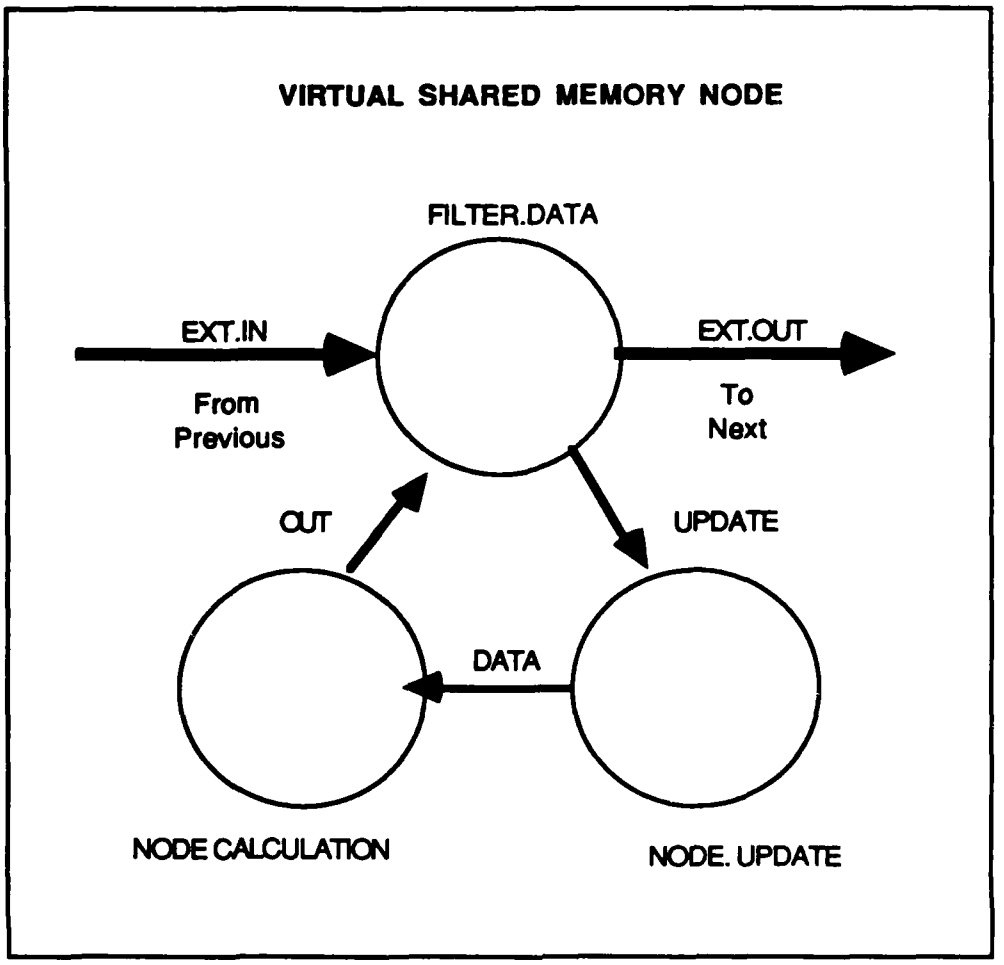


Figure 5.1

Node Activities

2. Assumptions

The design decisions discussed in this chapter are based on the following assumptions :

a. All nodes in the delay insertion loop are connected by DMA link engines with a 20 Megabit/second capacity.

b. The minimum size of the ring is four nodes.

c. The ring is incremented in multiples of four nodes.

d. The system is responsible for calculating a given global data structure for a particular class of problem domain. Each node is responsible for calculating a particular section of the system globally distributed data structure and distributing the resulting data throughout the system.

e. Implementation of the global data structure is accomplished by each node maintaining a replication of the data structure so that at any stage of system computation any node can provide the system state of computation.

f. A system state of computation is provided by monitoring the system eventcount status.

g. Total ordering of events throughout the system can be provided by sequencers.

h. Only one specified node is responsible for providing external system status monitoring. This is referred to as the Input/Output node (IO.node).

3. Process Description

a. Filter Data Process

This process is the delay insertion ring emulator. It is responsible for placing the node's updated data on the ring and removing messages the node placed on the ring. This process is the crux of the ring configuration. A major design decision in this process was a modification of the strict implementation of the delay insertion loop. A variable number of messages per node is permitted instead of the single message. This was implemented to permit determination of the optimum message passing method in the loop structure.

b. Update Process

Update process is responsible for updating the global data structure as the *Filter.data* process passes all system data to it. This includes its own data updates. This was a specific design

decision so that only the *Update* process could write to the data structure. The *Update* process, which monitors overall system status, synchronizes with and sends the appropriate values to *Calculate*.

c. Calculate Process

The *Calculate* process encapsulates the node calculation routine for providing updated data throughout the system. Each calculation provides the nodes updated data which is placed in the ring for distribution when the *Filter.data* process is ready to do so. This process is responsible for advancing the node's eventcount and issuing sequencer requests as appropriate.

4. Program Structure

The global data structure replicated within the node is considered an abstract data type with the operations within the update process, providing read and write operations on the data structure. Other abstract data types within the node are the eventcount for the node and any sequencers that may be required within the system. The OCCAM code for the node processes is shown in Figure 5.2.

C. MACRO-DESIGN DECISIONS

1. Process Priority

Examination of the program structure in Figure 5.2 shows the use of the high-priority process queue for message passing. This ensures that any message received is dispatched without delay to ensure good system performance. *Update* and *Calculate* are low-priority processes and will execute in round-robin fashion when no

no messages are to be dispatched. All processes which use the link engines should be run at high priority [At87].

```
NODE PROCESS

 INT event.count                          :
[node.no]INT eventcounters                :
[node.no * block.size]INT array           :
  CHAN OF ANY  ext.in, ext.out, int.in, int.out,  data :
PRI  PAR
       filter.data(ext.in,  int.in,  int.out,  ext.out)
    PAR
       update(int.out, data)
       calculate(data, int.out)
```

Figure 5.2

Node Process Listing

2. System Message Passing

Data communication is conducted by transmission of three discrete items: node identification, node data, and node data event count.

Each node receives these three items and either sends the message on or withdraws it from the ring. The present implementation presents the worst-case efficiency since three messages are dispatched and received for one data message produced by each node.

The alternative design is to package the three items in one data structure and send the data structure to each node. This would require an overhead of every message being encoded and decoded for each insertion and receipt at each node. These overheads would be

minimal to the overheads of 63 cycles per message incurred by each node for each communication set-up.

A basic two-tiered message passing scheme is implemented for the system to distinguish data packages and system coordination messages. Basically, any message with a non-negative integer header is system data, and negative headers are system manipulation calls such as sequencer requests and shut down calls.

Typing of channel protocol was not attempted for the construction of the ring, since it was not desirable for a development system which may need a flexible communication protocol. However, it is considered an important aspect to modularity in future development of such systems. For example, if for a given system a design decision was made concerning the message format and content, typed channels would provide valuable checks for module correctness and system compatibility at compile time. This would be important for non-homogeneous systems in modular development. For any subsequent work in this area, it is recommended that protocols be used after efficiency issues have been resolved and after adequate modular testing.

3. Synchronization and Data Passing Mechanism

The eventcount primitive await has not been implemented in this prototype. It is possible to implement an alternative to read by using the read primitive of dependent eventcounts together with the use of an internal channel to provide system synchronization. This is the case with the synchronization method between the *Update* and *Calculate* process.

This is not the only method of synchronization and internal node data passing. In certain instances, passing data by reference may be a more efficient method. This would require a sequential, vice the parallel, construct. Passing data by reference method was not used for this implementation because it was considered a restrictive method for system inter-node synchronization. The merits and examples of data passing methods are described well in [At87]; it is particularly suitable for certain data flow architectures.

4. System Shut Down

System shut down is achieved by passing two tokens. The first token informs all nodes to cease calculating and sending any further messages; the second shuts all system processes down.

The route taken by these tokens is shown in Figure 5.3. The criteria for system shutdown is flexible and may be generated either by a node in the loop or by the external system monitor.

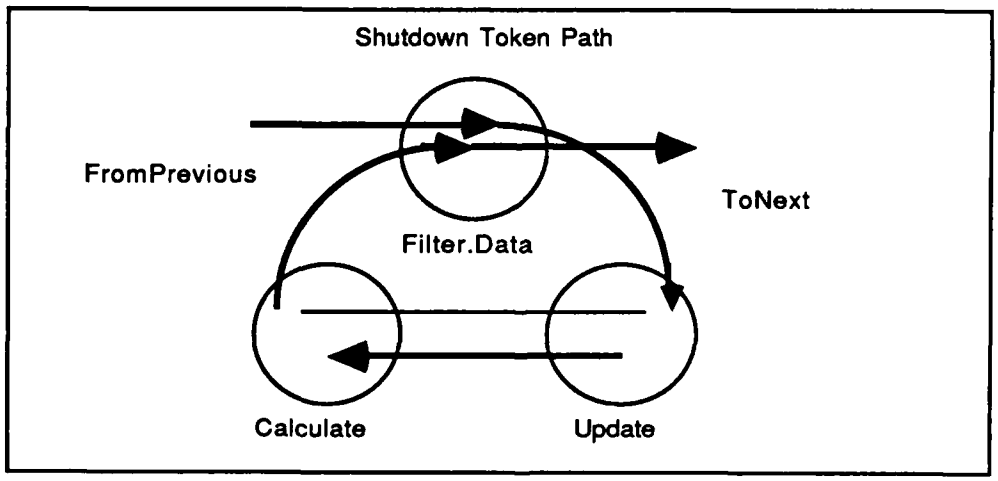


Figure 5.3

Shutdown Token Path

5. External System Monitoring

The state of the system is monitored by a process external to the ring structure. The IO node is responsible for tapping all messages to the external system monitor. The system monitor may then either display the system state as required or provide any necessary input data for the system operation. This process is the user interface to the system.

D. MICRO-DESIGN DECISIONS

1. Filter Process

The *Filter.data* process is solely responsible for the routing of messages throughout the system. Basically, the OCCAM language constructs PRI ALT/ALT easily allow multiplexing of data from several input channels. In *Filter.data*, there is one internal and one external channel to multiplex. The code for *Filter.data* is at Figure 5.4.

```
                    FILTER.DATA PROCESS
PROC Filter.Data(CHAN OF ANY External.In,  Internal.In,
                 Internal.Out, External.Out)
  ... PROC buffer
  ... PROC mix
  CHAN OF ANY own.data, other.data
  PAR
    Buffer(External.In,  other.data)
    Buffer(Internal.In, own.data)
    Mix(other.data, own.data, Internal.Out, External.Out)
:
```

Figure 5.4

Filter Data Process Listing

One very important rule for using link engines for extensive data routing processes is always use buffering to decouple the link

from the multiplexing process [At87, Pa87]. Failure to do so may result in deadlock and reduced efficiency. This occurred in a preliminary version of our implementation. It was found that without buffering, deadlocks occurred, especially as the number of nodes increased (as greater message traffic or number of messages allowed on the ring per node increased). Buffering the link engines allowed processes to run without any deadlock.

A corollary to the buffering rule is always decouple link engines from computation. This is a matter of efficiency, however, and not deadlock prevention. Decoupling link engines actually allows real concurrency of input, computation, and output.

2. Update

The main design issue is access to the globally distributed data structure. Only this process may access the data structure and send the appropriate data to *Calculate* for the necessary calculation. The two-tiered message passing scheme is supported throughout. Figure 5.5 shows the basic structure of the Update process.

3. Calculate

This process is responsible for the system calculation and event count advance primitive. The two-tiered message passing scheme prevails. Figure 5.6 shows the basic structure of the *Calculate* process.

```
PROC update(VAL INT machine, CHAN    OF ANY in, data)
 [block.size]INT sector                    :
 [node.no]INT event.count] event.  count   :
 INT node                                  :
 BOOL active                               :
 SEQ
  ... initialize variables
  WHILE active
    SEQ
        ... get message heade     r
        IF
          header positive
            ... get mes         sage data
            ... write da        ta to data structure
            ... determin        e synchronization details
          otherwise
            pass system t       oken
:
```

Figure 5.5

Update Listing

```
PROC calculate(VAL INT machine, CHAN OF ANY data, out)
 [block.size]INT sector     :
 INT event.count            :
 BOOL active                :
    ... PROC advance(event.count)
 ... PROC send.data(machine, sector, out)
 ... PROC heat.flow(left, right, length)
 SEQ
 ...    initialize
 ...    heat.flow(left, right, sector)
 ...   send.data(machine, sector, out)
 ...   advance(event.count)
 WHILE  active
    ... get synchronization message
    IF
      header positive
        ... get boundary conditions
          ...  heat.flow(left, right, sector)
         ...  send.data(machine, sector, out)
         ...  advance(event.count)
       otherwise
        pass system tokens
:
```

Figure 5.6

Calculate Listing

VI. EVALUATION OF THE VIRTUAL SHARED MEMORY IN A MULTI-TRANSPUTER NETWORK

A. OVERVIEW

The aim of this chapter is to examine the performance of the prototype virtual shared memory system in a multi-Transputer network. The prototype is evaluated using a representative problem which may arise using multi-processor architectures. The results are then compared to the ideal case and conclusions drawn from the results.

B. MULTI-PROCESSOR REPRESENTATIVE PROBLEMS

1. General

The heat flow problem was selected to evaluate the prototype virtual shared memory system since it is representative of many such problems that arise in meteorology, oceanography, engineering, and science. The single-dimensional heat flow solution was selected since it facilitated a simple template for a similar but more complicated problem domain.

2. The Heat Flow Problem

The heat flow problem in a single length of wire is described mathematically as a solution of the partial differential equation:

$$\frac{\partial u}{\partial t} := K\left(\frac{\partial u}{\partial x}\right)$$

with specified initial and boundary conditions. The problem is to examine the heat distribution in the wire as a function of time.

The system is responsible for determining the temperature of a particular length of wire. The length of wire is then divided into N sections which are directly mapped to the number of nodes in the network. These sections of wire are further divided into a number of P points which monitor temperature. This is shown in Figure 6.1.

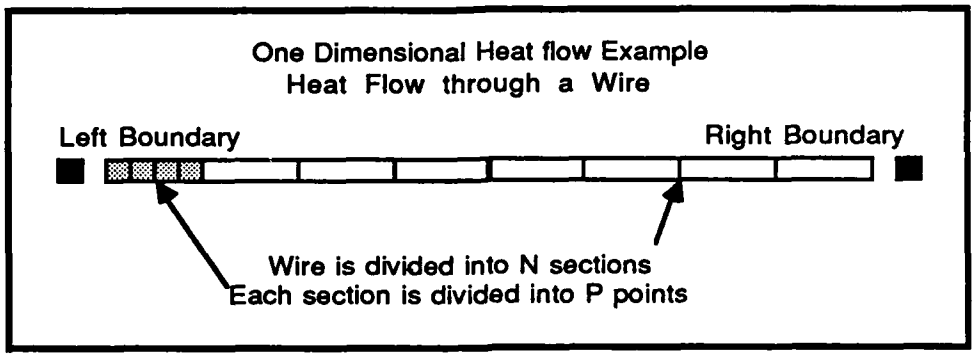


Figure 6.1

Heat Flow Through a Wire

The length of wire is represented by a globally distributed data structure which is a single dimensionalal array with (N * P) points. Each node is responsible for calculating the temperature of each of the P points in its a section of the wire.

The heat flow through the wire is computed by each node calculating the temperature of each point in its section and broadcasting it throughout the network. When all processors have completed the calculations, one iteration is said to have completed. This represents one unit of time. Iterative count is maintained by monitoring the eventcounts associated with each node in the system. For

example, when all eventcounts are at least equal to one indicates that the system has completed its first iteration.

Data for display is passed from the specialized IO.node in the network to a monitor process which displays the heat flow calculations periodically. This process is decoupled from the ring so the displaying of data incurs minimal degradation to the system. Source Code of the heat flow and the ring monitor process are available in Appendix C.

C. EVALUATION

1. Description

The prototype is evaluated using a four- and eight-node loop configuration allowing two messages per node in the loop. Global data structure sizes used are 100, 200, 400, 800, and 1600 integers. The network performance was timed over one thousand iterations. The timing was conducted from a monitoring process which timed the system from the passing of iteration information until the system stop token was received.

2. Results

Prototype results are given in Table 6.1 and Figure 6.2. Note that all performance results are measured for off-chip data for two reasons: it provides a worst-case evaluation and all results are under uniform conditions.

TABLE 6.1

PROTOTYPE PERFORMANCE RESULTS

Table 6.1 - Delay Insertion Loop Performance

Data Structure Size	Four Node	On/Off Chip	Eight Node	On/Off Chip
100	55,996	Off	111,106	Off
200	114,739	Off	230,134	Off
400	226,780	Off	448,324	Off
800	448,360	Off	896,921	Off
1600	892,628	Off	1763,736	Off

(Units in low level tick counts per 1000 iterations)

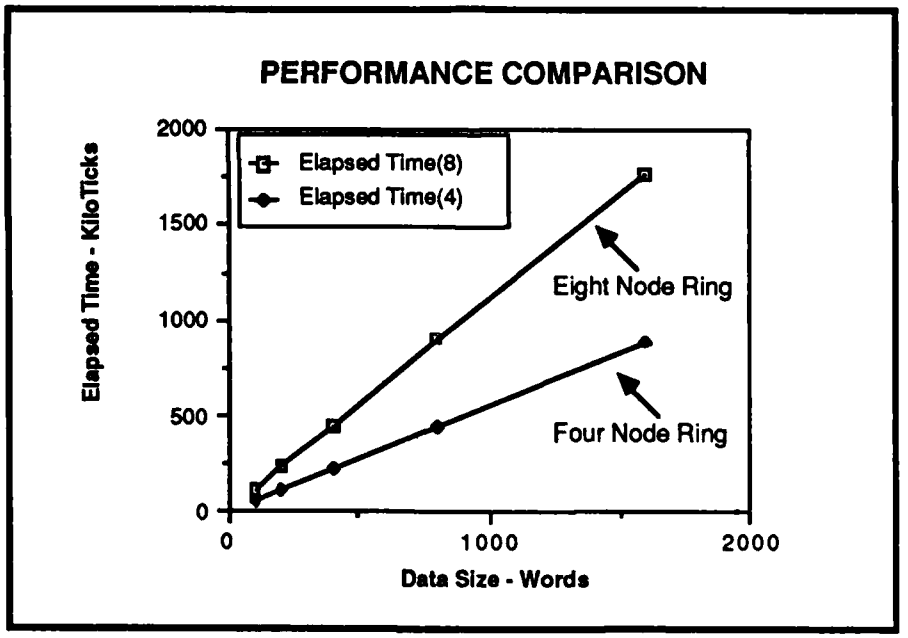


Figure 6.2

Performance Comparison

3. Observations

All system elapsed time data are plotted against data structure size per node. The slope of the four-node ring results is *exactly half* that of the eight-node results. This is a linear relationship. If the system throughput is thought of as the number of points calculated per unit time, then the throughput for all data on both four- and eight-node configurations remains relatively constant at an average of 7.1 points calculated per 1000 ticks (kiloticks). Thus, the ring configuration for this problem domain provides no linear performance improvement. This can be explained, however, by analyzing a time line of processor calculation and communication activities throughout the network.

Figure 6.3 shows an example of a full calculation and message cycles for a four node ring configuration. Each processor has two main activities, calculation and message passing. These two activities are shown in Figure 6.3. Each processor shares a link with an adjacent processor. For example, processor 0 and processor 1 share a link. It is assumed that each message is synchronized within some arbitrarily small time after it is sent. The heavy lines along the time axis represent processor cycle time used by each activity. Eack link activity is labelled with the originator of the message. Each processor calculation is labelled with the data set produced.

As each processor calculates its data, it is placed in the network for sequential distribution. Any calculation of data in a processor is known as processor useful activity. For strict implementation of the

delay insertion loop, only one message per node is allowed in the system at any time. This means that processor useful activity and idle time is dependent on message transit times through the ring configuration and length of processor useful activity. Examination of Figure 6.3 shows the length of the processor idle time. It is considerably longer than the useful activity time.

For the given problem domain of a linear heat distribution through a length of wire the calculation of each point in a section of wire may be described mathematically as:

$$U_i := \frac{1}{K}\left[U_{i-1} + \left(\frac{U_i}{K_0}\right) + U_{i+1}\right]$$

One may now estimate the calculation time for each point. The processor useful activity time may be calculated as a function of points per section of wire by calculating the execution time of the above equation. An approximate time of calculation for a point is 3 microseconds (3 microseconds per word). Message transmission time can be calculated as a function of link protocol and channel rate. [Va87] showed the net data transfer rate per T414 link is 23 bit times per byte or 4 microseconds per word. Therefore the processor spends more time in this problem sending data sets than calculating them. The network, therefore is said to be message bound and the idle time is dependent on the number of messages in the system.

4. Conclusions

The observations made show when a Transputer network is configured in a ring configuration and the problem domain is message

bound, system performance will not improve as processors are added to the system.

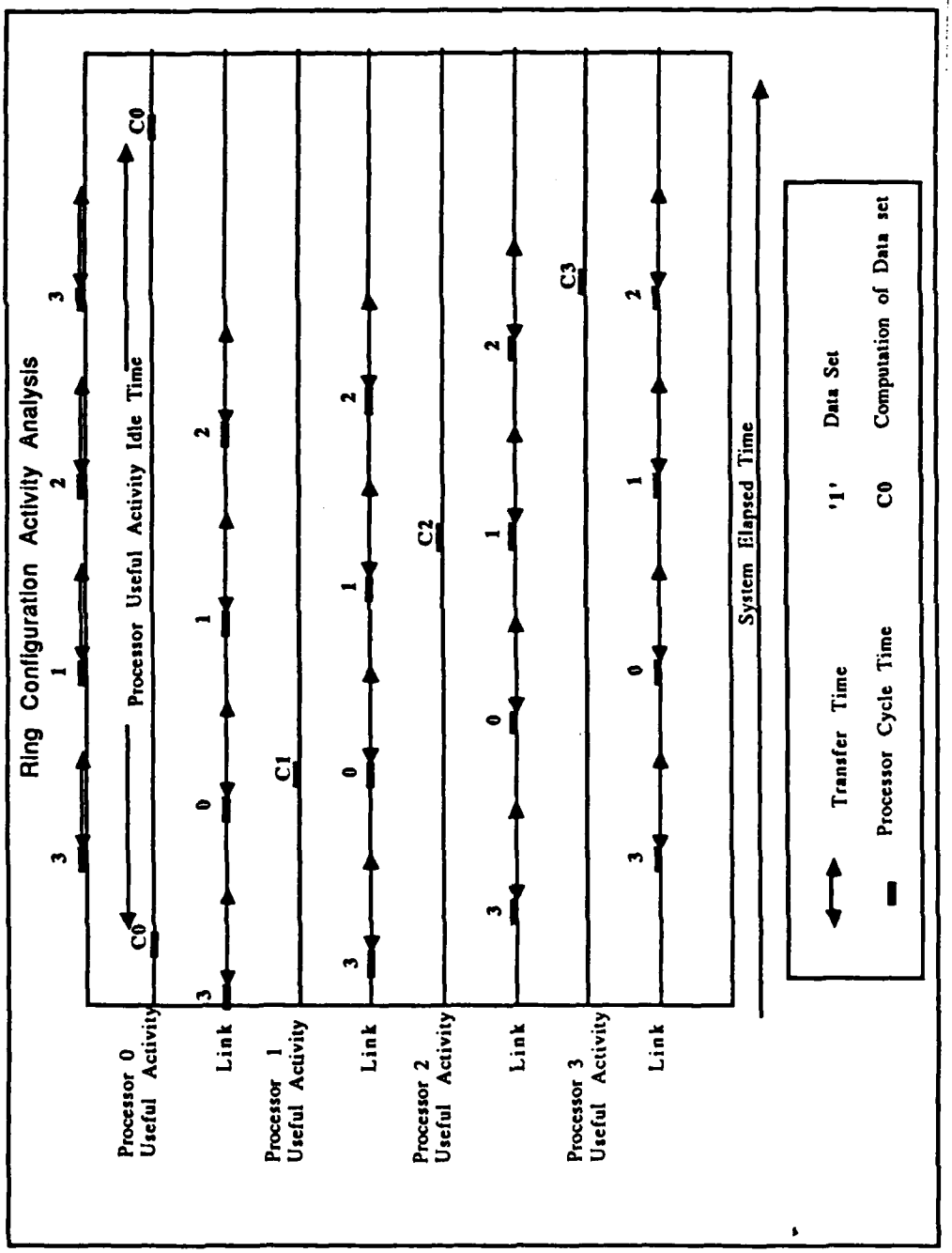


Figure 6.3

Ring Configuration Activity Analysis

Processor idle time is proportional to the number of nodes in the ring since message passing dominates calculation. The more processors waiting idly for data, the less effective the overall solution to the problem.

To ensure high system performance, one must ensure a high frequency of system useful activity or, conversely, reduce the processor idle time. This may be achieved by minimizing the message passing time or by ensuring each node's useful computation time is higher than the idle wait time. Message passing times may be reduced by passing essential data only throughout the network. For example, if the data set computation time in Figure 6.3 approached the message loop transit time, the idle time would be reduced producing more efficient system performance. Conversely, if in the single dimensionalal heat flow problem the boundary (essential) values only were passed then idle time would be reduced and overall system performance would improve.

In short, the single dimensional heat flow problem is not sufficiently computation intensive to test linear performance improvement of a ring configured Transputer network. Future work in this area requires a more comprehensive look at the problem domain computation versus message passing time ratio.

VII. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This thesis has investigated and documented some very fundamental issues involving programming the Transputer and has investigated its suitability as a future weapon system processor. The topics selected for discussion are germane to network configuration and of course do not cover all details. An attempt has been made to include as many suitable references as possible for the reader for further discussion and examples. The topics covered in this thesis are only a preliminary investigation into this new frontier of microprocessors.

Unless the basic notions of the Transputer model are revealed, further investigation may provide misleading results. The first half of this thesis has attempted to distill the essence of the basic hardware implementation. Understanding this aspect should give a better insight into improving network performance.

Another fundamental issue investigated concerning networks of Transputers is the maximum degradation on a Transputer CPU when all link engines are operating. The major overhead is setting up the data transfer of 21 cycles per message. The other overhead is due to cycle stealing on the internal bus by each of the link engines as they transfer data. The maximum degradation was calculated to be 25% for the T414 link protocol. Predictions of degradation for a particular instruction did not prove conclusive. Message passing in Transputer

networks should be packaged into long essential messages. If the maximum degradation of the Transputer CPU due to link engine cycle stealing is 25% for a 10 MIP processor, then the overall system performance is still very satisfactory.

Timing in the Transputer must be done with great care. Naive use of the TIMER will produce misleading results. The programmer of real-time programs must take great care to ensure correct program modelling when using the TIMER.

The major conclusion that one could make is that programming Transputer networks requires a detailed knowledge of how the hardware implementation works before the full performance can be harnessed. The aim of the implementation of a virtual shared memory is to use the link engines and the CPU to the maximum with minimal mutual interference. Results obtained from the evaluation indicate that to implement virtual shared memory processor useful activity must be analyzed against message-passing time. Message bound systems do not provide linear performance gains in a ring configuration.

B. RECOMMENDATIONS

The emphasis, however, in the AEGIS modelling group is to find a suitable architecture for future weapon systems control. The MCORTEX system has proved to be a satisfactory system using hierarchical shared memory and bus systems. The aim is to map the Transputer to such a system, optimizing the unique Transputer architecture. This thesis is the first to implement and evaluate such an

gramming such a network and the understanding of the hardware implementation. To this aim, it is recommended that the virtual shared memory prototype be further investigated for suitable improvement and rigorous evaluation for computation bound problems. The major unresolved problem from this research is the prediction of degradation in the performance of the Transputer CPU caused by the links engines' activity. It is recommended that this be further investigated and documented.

Programming productivity is enhanced by using a wide variety of tools. The latest edition of the Transputer Development System provides an adequate debugger for debugging networks of Transputers which should improve program productivity. The library system included in the latest OCCAM language compiler has provided some excellent routines for use. It is recommended that further investigation be made into the full utilities of the Transputer Development System for use. This includes such items as making bootable files for stand-alone application programs and incrementally improving the INMOS supplied libraries. Detailed investigation into the tool set available will enhance further activities and research with the Transputer networks. Further, it is recommended that languages such as C and Pascal and ADA (when it becomes available) be investigated for use. The use of these languages may improve programming productivity by allowing program portability and the use of language features not yet available in OCCAM.

APPENDIX A

PROCESSOR PERFORMANCE DEGRADATION EVALUATION SOURCE CODE

A. SUMMARY

The aim of this evaluation is to measure the performance degradation of the Transputer CPU while all eight link engines are performing data transfer. The logical structure of the program is shown in Figure A.1. The center node contains the evaluation code. Each satellite contains message-passing code to work the link engines. One satellite contains the user interface to report the results.

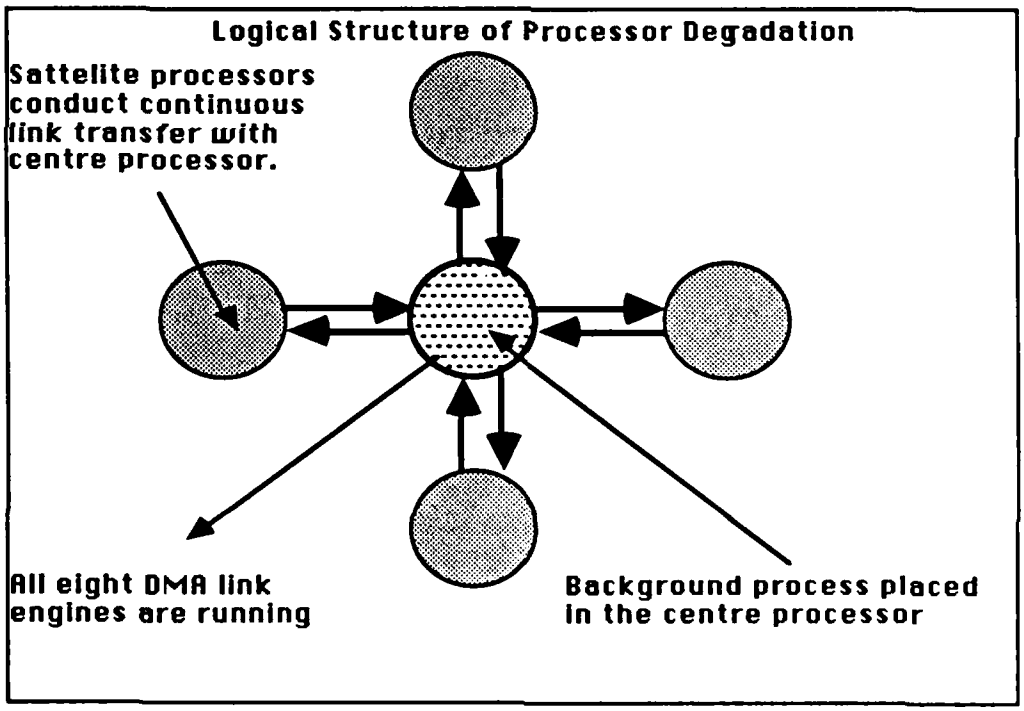


Figure A.1

Logical Structure of Processor Degradation Program

B. SOURCE CODE

```
CHAN OF ANY in.1, out.1 :
CHAN OF ANY in.2, out.2 :
CHAN OF ANY in.3, out.3 :

PLACED PAR
  PROCESSOR 0 T4
  PLACE in.1 AT linkIn2 :
  PLACE in.2 AT linkIn1 :
  PLACE in.3 AT linkIn3 :
  PLACE out.1 AT linkOut2 :
  PLACE out.2 AT linkOut1 :
  PLACE out.3 AT linkOut3 :
  central.node(0, in.1, in.2, in.3, out.1, out.2, out.3)

PLACED PAR
  PROCESSOR 1 T4
    PLACE in.1  AT linkOut3  :
    PLACE out.1 AT linkIn3 :
    busy.transfer.T1(1, out.1, in.1)
  PROCESSOR 2 T4
    PLACE in.2  AT linkOut0  :
    PLACE out.2 AT linkIn0 :
    busy.transfer.T1(2, out.2, in.2)
  PROCESSOR 3 T4
    PLACE in.3  AT linkOut2  :
    PLACE out.3 AT linkIn2 :
    busy.transfer.T1(3, out.3, in.3)
```

```
PROC central.node(VAL INT machine,
                  CHAN OF ANY in.1, in.2, in.3,
                              out.1, out.2, out.3)
  VAL data.size IS 1024 :
  -- this size was chosen to fit all data on chip
  VAL par.tag IS -1 :
  -- connected to host user interface
  CHAN OF ANY in.0, out.0 :
  PLACE in.0 AT 4 :
  PLACE out.0 AT 0 :

  [4]CHAN OF ANY to :
  TIMER clock :
  INT CT :
  [2]INT start, stop :
  INT link.iteration, length :
  INT x1, x2, x3, z1, z2, z3 :
  [data.size]INT data0, data1, data2, data3 :
  [data.size/4]INT sector :

  SEQ
    -- initialize
    SEQ i = 0 FOR data.size/4
      sector[i] := 100
    -- synchronize data
    in.0 ? link.iteration
    in.0 ? CT
    in.0 ? length
    out.0 ! pripar.tag
    -- unhindered computation timing
    INT now :
    SEQ
      now := 10
      clock ? start[0]
      -- block.move code
      SEQ i = 0 FOR CT
      [data0 FROM 0 FOR length ] = [sector FROM 0 FOR length]
      -- division code
      SEQ i = 0 FOR CT
      now := now / 1
      clock ? stop[0]
      -- synchronize satellites
      PAR
        out.1 ! link.iteration , length
        out.2 ! link.iteration , length
        out.3 ! link.iteration , length
```

```
-- start all link engines
PRI PAR
  PAR
    PAR
      SEQ i = 0 FOR link.iteration
        SEQ
        in.0   ? x1; [data0 FROM 0 FOR length]; z1
        to[0]  ! x1; [data0 FROM 0 FOR length]; z1
      SEQ i = 0 FOR link.iteration
        SEQ
        to[0]  ? x3; [data3 FROM 0 FOR length]; z3
        out.0  ! x3; [data3 FROM 0 FOR length]; z3
    PAR
      SEQ i = 0 FOR link.iteration
        SEQ
        in.1   ? x1; [data1 FROM 0 FOR length]; z1
        to[1]  ! x1; [data1 FROM 0 FOR length]; z1
      SEQ i = 0 FOR link.iteration
        SEQ
        to[1]  ? x3; [data3 FROM 0 FOR length]; z3
        out.1  ! x3; [data3 FROM 0 FOR length]; z3

    PAR
      SEQ i = 0 FOR link.iteration
        SEQ
        in.2   ? x1; [data1 FROM 0 FOR length]; z1
        to[2]  ! x1; [data1 FROM 0 FOR length]; z1
      SEQ i = 0 FOR link.iteration
        SEQ
        to[2]  ? x3; [data3 FROM 0 FOR length]; z3
        out.2  ! x3; [data3 FROM 0 FOR length]; z3
  PAR
    SEQ i = 0 FOR link.iteration
      SEQ
      in.3   ? x1; [data1 FROM 0 FOR length]; z1
      to[3]  ! x1; [data1 FROM 0 FOR length]; z1
    SEQ i = 0 FOR link.iteration
      SEQ
      to[3]  ? x3; [data3 FROM 0 FOR length]; z3
      out.3  ! x3; [data3 FROM 0 FOR length]; z3
```

```
        SEQ
          deschedule(10)
          SEQ k = 0 FOR link.iteration
              INT now :
              SEQ
                now := 10
                -- interfered computation timing
                clock ? start[1]
                SEQ i = 0 FOR CT
                    [data0 FROM 0 FOR length ] := [sector FROM 0 FOR length]
                clock ?  stop[1]
          -- send results to display
          out.0 ! (stop[1] - start[1]) ; (stop[0] - start[0])
:
```

```
PROC busy.transfer.T1(VAL INT machine, CHAN OF ANY in, out)
    VAL data.size IS 1024          :
    INT x1, x2, z1, z2             :
    INT link.iteration, length     :
    [data.size]INT data.out, data.out :
        :
    SEQ
      -- initialize
      SEQ i = 0 FOR data.size
          data.out[i] := machine
      x1 := machine
      z1 := 1
      -- synchronize data
      in ? link.iteration; length
      -- send and receive
      PAR
          SEQ
            SEQ i = 0 FOR link.iteration
                out ! x1; [data.out FROM 0 FOR length] ; z1
          SEQ
            SEQ k = 0 FOR link.iteration
                out ! x2; [data.in FROM 0 FOR length] ; z2
```

APPENDIX B

UNIDIRECTIONAL LOOP CONFIGURATION SOURCE CODE

```
--------------------------------------------------------------------------------
-- N node uni-directional ring configured for B003 application
--------------------------------------------------------------------------------
AUTHOR          :   SJ HART
DATE            :                                              25
OCTOBER 1987
VERSION         :   2.0
ENVIRONMENT     :   MACINTOSH 512 TDS 2.0 BETA 2.0(MARCH 1987)
FILE.NAME       :   ringstructure.TSR
TOP.FILE        :   TEST.TOP
DESCRIPTION     :   Uni directional ring structure
--------------------------------------------------------------------------------

-- link channel offsets

VAL link0in  IS 4    :
VAL link1in  IS 5    :
VAL link2in  IS 6    :
VAL link3in  IS 7    :
VAL link0out IS 0    :
VAL link1out IS 1    :
VAL link2out IS 2    :
VAL link3out IS 3    :

-- Each internal channel is associated with a table indexed
-- when the internal channel is mapped onto an external channel

VAL clockwise.in    IS [link1in, link3in,  link3in,  link3in] :
VAL clockwise.out   IS [link2out, link2out, link2out, link0out]:

-- this varies according to network size

VAL No.B003 IS 1          :
VAL n       IS (4 * No.B003) :
VAL node.no IS n          :

-- channel declaration
```

```
[node.no] CHAN OF ANY  clockwise  :

-- separately compiled "node" to be extracted to all nodes

... SC modules

-- Configuration Code

-- MACHINE IS THE NODE REPLICATOR IDENTIFIER

PLACED PAR
  VAL machine IS 0      :
  PROCESSOR machine T4
   -- position of node within the B003 board (0..3)
       VAL clock.in          IS (machine + (node.no-1)) \ node.no      :
       VAL clock.out         IS  machine                               :
       VAL map.index         IS  machine \ 4                           :
       CHAN OF ANY from.kb, to.monitor                                 :
       PLACE clockwise[clock.in]    AT clockwise.in      [map.index]   :
       PLACE clockwise[clock.out]   AT clockwise.out     [map.index]   :
       PLACE to.monitor   AT Link0in                                   :
       PLACE from.kb      AT Link0out                                  :

       O.node02(machine, from.kb, clockwise[clock.in],
                        clockwise[clock.out], to.monitor)

  PLACED PAR machine = 1 FOR node.no-1
    PROCESSOR machine T4

       -- position the  node within the B003 board (0..3)

       VAL clock.in  IS (machine + (node.no-1)) \ node.no              :
       VAL clock.out IS  machine                                       :
       VAL map.index     IS  machine \4                                :
       PLACE clockwise[clock.in]    AT clockwise.in[map.index]         :
       PLACE clockwise[clock.out] AT clockwise.out[map.index]          :

       node( machine,  clockwise [clock.in], clockwise [clock.out])
```

APPENDIX C

IMPLEMENTATION OF VIRTUAL SHARED MEMORY SOURCE CODE

A. SUMMARY

The following paragraph summarises the processes contained in the implementation of a virtual shared memory in a network of Transputers. The code of each of these processes follows.

```
--------------------------------------------------------------------------------
PROC node(VAL INT machine, CHAN OF ANY ext.in, ext.out)
--------------------------------------------------------------------------------
-- This process is the code contained in all nodes throughout the network.
-- The process contains three parallel processes
-- (1) filter.data
-- (2) update
-- (3) calculate
  ------------------------------------------------------------------------------
  PROC buffer(CHAN OF ANY in, out)
  ------------------------------------------------------------------------------
  -- single software buffer for prototype virtual shared memory system.
  ------------------------------------------------------------------------------
  PROC mix(CHAN OF ANY ext.in, int.in, ext.out, int.out)
  ------------------------------------------------------------------------------
  -- This process multiplexes two channels.
  -- One external and one internal.  This process is
  -- responsible for the network message passing scheme.
  ------------------------------------------------------------------------------
  PROC filter.data(VAL INT machine,              -- node
                   CHAN OF ANY ext.in, int.in,   -- in
                               int.out, ext.out) -- out
  ------------------------------------------------------------------------------
  -- To be used by receive process for multiplexing data from
```

```
    -- previous node OR the node.calculation
--------------------------------------------------------------------------------
PROC update(VAL INT machine, CHAN OF ANY in, data)
--------------------------------------------------------------------------------
  -- Place sector in the virtual array according to the node
  -- from whence it came and send synchronization data to calculate
-------------------------------------------------------------
    PROC write.data(VAL INT start.point,
                    VAL [ ]INT sector, [ ]INT array )
-------------------------------------------------------------
--------------------------------------------------------------------------------
PROC calculate(VAL INT machine, CHAN OF ANY  data,  out)
--------------------------------------------------------------------------------
  -- this procedure will dispatch all details
  -- concerning the nodes calculations and update the event count
  -- the process assume synchronization data from update via data channel
    ---------------------------
    PROC advance(INT event.count)
    ---------------------------
    -- advances the given eventcount by one
    ------------------------------------------------------------------------
    PROC send.data(VAL INT machine, [ ]INT sector, CHAN OF ANY  out)
    ------------------------------------------------------------------------
    -- Implements communication protocol for the prototype
    ---------------------------------------
    PROC heat.flow(VAL INT left, right,  [ ]INT length)
    ---------------------------------------
     -- One dimensional heat flow calculation
     -- This is typical of probelms that may be solved in this network
--------------------------------------------------------------------------------
PROC IO.node(VAL INT machine,
             CHAN OF ANY from.kb, ext.in, ext.out, to.screen)
--------------------------------------------------------------------------------
-- Display the node data structure at each consistent data point to
-- external monitor
    ------------------------------------------------------------
    PROC IO.filter.MUX(VAL INT machine,                    -- node
```

```
                CHAN OF     ANY ext.in, int.in,                 -- in
                        int.out, ext.out, to.display) -- out
  ---------------------------------------------------------------
  ---------------------------------------------------------------
  PROC IO.update(VAL INT machine, CHAN OF ANY in, data)
  ---------------------------------------------------------------
-----------------------------------------------------
PROC ring.monitor(CHAN OF ANY keyboard, screen)
-----------------------------------------------------
  -- host machine user interface with the network
  ---------------------------------------------------------------
  PROC monitor( CHAN OF ANY from.kb, to.monitor, to.screen,
                                  from.monitor)
  ---------------------------------------------------------------
    -- external monitoring process to the system
    -------------------------------------------------------------
    PROC write.int(CHAN OF ANY to.screen, VAL INT number, field)
    -------------------------------------------------------------
    -- display utility for numeric data  to  screen

    ---------------------------------------
    PROC clear.line(CHAN OF ANY to.screen)
    ---------------------------------------
    -- utility for clearing line
    -----------------------------------------------
    PROC go.to(CHAN OF ANY to.screen, VAL INT X, Y)
    -----------------------------------------------
    -- utilty for cursor position
    -------------------------------------------------
    PROC write.s(CHAN OF ANY to.screen, VAL [] BYTE string)
    -------------------------------------------------
    ---------------------------------------------------------
    PROC collect.data(CHAN OF ANY  to.monitor, to.data.structure)
    ---------------------------------------------------------
    -- multi buffering process only to decouple display from the system
    ---------------------------------------------------------
```

```
PROC update.memory(CHAN OF ANY in,  to.screen, []INT array
                                    [node.no]INT event.count)
```

```
-- receives the two tiered messages from the system and
-- responds accordingly. Primarily responsible for timings and data
```

B. DETAILED SOURCE CODE

```
PROC node(VAL INT machine, CHAN OF ANY ext.in, ext.out)
-- This process is the code contained in all nodes throughout the network.
-- The process contains three parallel processes
-- (1) filter.data
-- (2) update
-- (3) calculate
-----------------------------------------------------
 -- node variables
-----------------------------------------------------
 [500]INT on.chip.space : -- push all data off-chip
 -- internal channels
   CHAN OF ANY int.in, int.out, data  :

 -- system variables
   VAL node.no          IS   4        :
   VAL block.size       IS 100        :
   [node.no*block.size]INT array      :  -- node data structure 1D array

 -- system tokens
   VAL stop.token       IS -1         :
   VAL shut.down.token IS -2          :

   VAL otherwise IS TRUE              :

   PRI PAR
      filter.data(machine, ext.in, int.out, int.in, ext.out)
         PAR
            update(machine, int.in, data)
            calculate  (machine, data, int.out)
:
```

```
PROC buffer(CHAN OF ANY in, out)
  -- single software buffer for prototype virtual shared memory system.
:
INT node, event.count :
  [block.size]INT data :
  BOOL active :
  SEQ
  active := TRUE
  WHILE active
    SEQ
      in ? node
      IF
        node >= 0
          SEQ
            in  ? data ; event.count
            out  ! node; data; event.count
        otherwise
          IF
            node = shut.down.token
              SEQ
                out ! node
                active := FALSE
            node = stop.token
              out ! node
            otherwise
              SKIP
:
```

```
PROC mix(CHAN OF ANY ext.in, int.in, ext.out, int.out)
 -- This process multiplexes two channels.
 -- One external and one internal.  This process is
 -- responsible for the network message passing scheme.
-------------------------------------------------------------------

    VAL max.message.load IS 1   :
    INT node , event.count      :
    INT message.no              :
    [block.size]INT sector      :
    BOOL active                 :
    SEQ
    -- initialization
    event.count := 0
    active := TRUE
    message.no := 0
    WHILE active
        PRI ALT
            ---------------------------------------------------
                (message.no < max.message.load)  & int.in ? node
                -- internal input from calculation
            ---------------------------------------------------
            SEQ
                IF
                    node >= 0
                        SEQ
                            int.in ? sector ; event.count
                            PAR    -- send the update to next node
                                ext.out   !  node ; sector; event.count
                                int.out   !  node ; sector; event.count
                            message.no := message.no + 1
                    otherwise      -- node < 0
                    IF
                        node = stop.token
                            ext.out ! stop.token  -- dispatches stop.token
                        node = shut.down.token
                            SEQ
                                ext.out ! shut.down.token
                                active := FALSE  -- dispatch then shut down

            ---------------------------------------------------
            ext.in  ? node
            -- external input from previous node
             ---------------------------------------------------
            SEQ
```

```
IF
    node <> machine
        -- includes the stop.tokens and other nodes
        SEQ
            IF
                node >= 0     -- send on & stop process
                    SEQ
                        ext.in ? sector ; event.count
                        PAR
                            ext.out   ! node ; sector ; event.count
                            int.out   ! node ; sector ; event.count
                otherwise  -- node < 0
                    SEQ
                        IF
                            node = stop.token
                                int.out   ! stop.token
                            -- the stop.token has travelled
                            -- the full ring and stopped ALL processes
                            node = shut.down.token
                                int.out   ! shut.down.token
                            -- shut down token will shut down all
                            -- but the IO node
                            otherwise
                                SKIP
        node =  machine
            SEQ
            ext.in ? sector ; event.count
            message.no := message.no  - 1
        otherwise
            SKIP
:
```

```
PROC filter.data(VAL INT machine,                -- node
                 CHAN OF ANY ext.in, int.in,     -- in
                             int.out, ext.out)   -- out
-- To be used by receive process for multiplexing data from
-- previous node OR the node.calculation
--------------------------------------------------------------

  CHAN OF ANY other.data, my data
  PAR
    buffer(ext.in, other.data)
    buffer(int.in, my.data)
    mix(other.data, my.data, ext out, int out)
:
```

```
PROC update(VAL INT machine, CHAN OF ANY in, data)
  -- Place sector in the virtual array according to the node
  -- from whence it came and send synchronization data to calculate
  VAL INT initial.value IS 0    :
  INT start.point, count        :
  [block.size]INT   sector      : -- node responsibility
  [node.no]INT event.count      : -- iteration record
  INT  node                     : -- which machine
  BOOL active                   :
  PROC write.data(VAL INT start.point,
                  VAL [ ]INT sector, [ ]INT array )
    SEQ
    [array FROM start.point FOR block.size ] := sector
  SEQ
    -- initialize variables
    SEQ i= 0 FOR block.size
      sector[i] := -machine
    SEQ k = 0 FOR (node.no * block.size)
      array[k] := initial.value
    active := TRUE
    WHILE active
    SEQ
      in ? node
        -- two-tier message system
        IF
          node >= 0
            SEQ
              in ? sector ; event.count[node]
              start.point := node * block.size
              write.data(start.point, sector, array)
              -- synchronize the calculation
              IF
                node = ((machine + (node.no - 1))\(node.no))
                  VAL right.boundary IS   0 (INT)   :
                  VAL left.boundary  IS 10000 (INT)   :
                  INT left, right           :
                  SEQ
```

```
                        -- determine boundary conditions
                        IF

                            machine = 0
                              SEQ
                                 left  := left.boundary
                                 right := array[block.size + 1]

                            machine = (node.no - 1)
                              SEQ
                                 left  := array[((block.size * machine) - 1)]
                                 right := right.boundary

                            otherwise
                              SEQ
                                 left  := array[((block.size * machine) -1)]
                                 right := array[(block.size * (machine +1))]
                                data ! node ; left ; right

                    otherwise
                       SKIP
otherwise  -- node < 0
-- pass the system message through the node
    IF
        node = stop.token
            data ! stop.token
        node = shut.down.token
            SEQ
                data ! shut.down.token
                active := FALSE
        otherwise
            SKIP
```

```
--------------------------------------------------------------------------------
PROC calculate(VAL INT machine, CHAN OF ANY  data,  out)
-- this procedure will dispatch all details
-- concerning the nodes calculations and update the event count
-- the process assume synchronization data from update via data channel
--------------------------------------------------------------------------------
VAL INT initial.value    IS      0 (INT)    :
VAL right.boundary       IS      0 (INT)    :
VAL left.boundary        IS 10000 (INT)     :
INT left, right                             :
[block.size]INT sector                      :
BOOL active, stop.signal                    :
INT event.count                             :
--------------------------------------
PROC advance(INT event.count)
--------------------------------------
  SEQ
    event.count := event.count + 1
:
--------------------------------------------------------------------------------
PROC send.data(VAL INT machine, [ ]INT sector, CHAN OF ANY  out)
--------------------------------------------------------------------------------
-- Implements communication protocol for the prototype
    SEQ
    out ! machine ; sector ; event.count
:
--------------------------------------------------------------------------------
PROC heat.flow(VAL INT left, right,  [ ]INT length)
 --- one dimensional heat flow calculation
-- This is typical of probelms that may be solved in this network
--------------------------------------------------------------------------------
  VAL rate  IS 1   (INT)   :
    SEQ
    length[0] := ((left + (rate*length[0])) + length[1])/(rate+2)
    SEQ i = 1 FOR (block.size - 2)
      length[i] := ((length[i-1]  + (rate*length[i])) + length[i+1])/(rate+2)
      length[block.size-1] :=  ((length[block.size-2] +
            (rate * length[block.size-1])) + right) / (rate+2)
:
```

```
SEQ
  -- initialization
  SEQ i = 0 FOR  block.size
    sector[i] := initial.value
  SEQ k = 0 FOR node.no * block.size
    array[k] := initial.value
  active := TRUE
  stop.signal := FALSE
  event.count := 0

  --- a simple calculation
  IF
    machine = 0
      SEQ
        left  := left.boundary
        right := array[block.size + 1]
    machine = (node.no - 1)
      SEQ
        left  := array[((block.size * machine) - 1)]
        right := right.boundary
    otherwise
      SEQ
        left  := array[((block.size * machine) -1)]
        right := array[(block.size * (machine + 1))]
        heat.flow(left, right, sector)
        send.data(machine, sector, out)
        advance(event.count)
    WHILE active
      INT node :
      SEQ
        data ? node          -- synchronise or stop
        IF
          node >= 0          -- filter code for negative numbers
            IF
              stop.signal
                SKIP
              otherwise
                SEQ
                  data ? left; right
                  -- get synchronization data
                  heat.flow(left, right, sector)
                  send.data(machine, sector, out)
                  advance (event.count)


          otherwise
            SEQ
```

```
-- system messages
IF
    node = stop token
      SEQ
        out ! stop token
        stop signal = FALSE
        -- do not send any more data
    node = shut down token
      SEQ
        out ! shut down token          shut down
        active = FALSE
    otherwise
      SKIP
```

```
PROC IO.node(VAL INT machine,
             CHAN OF ANY from.kb, ext.in, ext.out, to.screen)
-- Display the node data structure at each consistent data point to
-- external monitor

  [500]INT on.chip.space

  CHAN OF ANY int.in, int.out, data, to update
  CHAN OF ANY other.data, my data
  VAL stop.token          IS -1 (INT)
  VAL shut.down.token     IS -2 (INT)
  VAL time.token          IS -3 (INT)
  VAL otherwise IS TRUE
  VAL node.no   IS 4
  VAL block.size IS 100
  INT iteration
  [node.no*block.size]INT array                    -- node data structure
  SEQ
      from.kb  ? iteration
      to.screen ! iteration
      PRI PAR
        IO.filter.MUX(machine, ext.in, int.out, int.in, ext.out, to.screen)
        PAR
          IO.update(machine, int.in , data)
          calculate  (machine, data, int.out)
```

```
--------------------------------------------------------------------------
PROC IO.filter.MUX(VAL INT machine,                    -- node
                   CHAN OF ANY ext.in, int.in,         -- in
                               int.out, ext.out, to.display)  -- out
--------------------------------------------------------------------------
  VAL max.message.load IS 1          :
  CHAN OF ANY other.data, my.data    :
  [max.message.load+1]CHAN OF ANY extension :
  PAR
    buffer(ext.in, other.data)
    buffer(int.in, my.data)
    IO.mix(other.data, my.data, ext.out, int.out, to.display)
:
--------------------------------------------------------------------------
PROC IO.mix(CHAN OF ANY ext.in, int.in,
                        ext.out, int.out, to.display)
--------------------------------------------------------------------------
    TIMER clock                 : -- timer for message circuit
    INT start, stop, message.no:
    INT node, event.count       :
    [block.size]INT sector      :
    BOOL active                 :
    SEQ
      -- initialize
      event.count := 0
      active := TRUE
      message.no := 0
      WHILE active
        PRI ALT
          ------------------------------------------------------
          (message.no < max.message.load) & int.in ? node
            -- internal input from calculation
          ------------------------------------------------------
          SEQ
            IF
              node >= 0
                SEQ
                  int.in ? sector ; event.count
                  PAR   -- send the update to next node
                    ext.out    ! node ; sector; event.count
                    int.out    ! node ; sector; event.count
                    to.display ! node ; sector; event.count
                  message.no := message.no + 1
                  clock ? start
```

```
                otherwise          -- calc.in node < 0
                    IF  -- check if the stop token and flush the system
                        node = stop.token
                            PAR
                            ext.out  ! stop.token --dispatches stop.token
                             to.display ! stop.token
                        node = shut.down.token
                            SEQ
                                to.display ! shut.down.token
                                active := FALSE
                              -- shut down last of PROC's
------------------------------------------------------
ext.in  ? node     -- external input from previous node
------------------------------------------------------
    IF
        node <> machine  -- includes the stop.tokens
            node >= 0
                SEQ
                    ext.in ? sector ; event.count
                    PAR
                    ext.out     ! node ; sector ; event.count
                    int.out     ! node ; sector ; event.count
                    to.display ! node ; sector ; event.count
                    IF
                        node = stop.token
                          PAR
                              ext.out    ! shut.down.token
                              to.display ! stop.token
                               -- the stop.token has traveled the full ring
                        node = shut.down.token
                          SEQ
                              to.display  ! time.token ; (stop - start)
                              int.out     !  shut.down.token
                                -- shut down token has shut down
                                --all but the IO node
                        otherwise
                            SKIP
                node =  machine
                    SEQ
                        ext.in ? sector ; event.count
                        clock ? stop -- stop loop message timing
                        message.no := message.no - 1
```

```
PROC IO.update(VAL INT machine, CHAN OF ANY in, data)

  VAL INT initial.value IS 0  :
  INT start.point, count      :
  [block.size]INT  sector     : -- node responsibility
  [node.no]INT event.count    : -- iteration record
  INT  node                   : -- which machine
  BOOL active, stop.set       :
  SEQ
    SEQ i = 0 FOR block.size
      sector[i] := -machine
    SEQ k = 0 FOR (node.no*block.size)
      array[k] := initial.value
    active := TRUE
    stop.set := FALSE
    WHILE active
      SEQ
        in ? node
        IF
          node >= 0
            SEQ
              in ? sector ; event.count[node]
              start.point := node * block.size
              write.data(start.point, sector, array)
              -- Stop conditional section
              IF
                IF i = 0 FOR node.no-1
                  (event.count[i] <= iteration) OR stop.set
                    SKIP
                otherwise
                  SEQ
                    stop.set := TRUE
                    data ! stop.token
              IF
                ((node = ((machine+(node.no-1))\node.no))
                AND (NOT stop.set))
                  VAL left.boundary  IS 10000 (INT) :
                  VAL right.boundary IS     0 (INT) :
                  INT left, right :
                  SEQ
                  IF
                    machine = 0
                      SEQ
                        left  := left.boundary
```

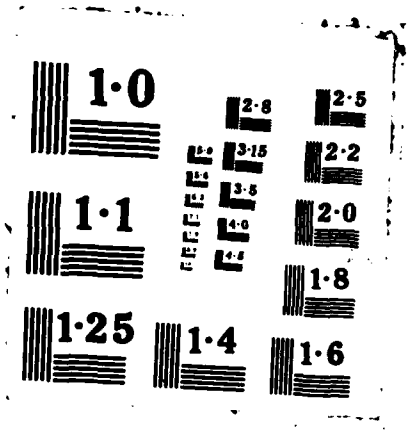
```
                        right := array[block.size + 1]
                    machine = (node.no - 1)
                      SEQ
                        left  := array[((block.size * machine) - 1)
                        right := right.boundary
                    otherwise
                      SEQ
                          left  := array[((block.size * machine) -1)]
                          right := array[(block.size * (machine+1))]
                        data ! node; left; right
                otherwise
                    SKIP
        otherwise  -- node < 0
            IF
                node = stop.token
                    data ! stop.token
                node = shut.down.token
                    SEQ
                        data ! shut.down.token
                        active := FALSE
                otherwise
                    SKIP
```

```
PROC Ring.Monitor(CHAN OF ANY keyboard, screen)

  CHAN OF ANY to.B003, from.B003 :
  PLACE to.B003   AT 2 : -- link 2 out
  PLACE from.B003 AT 6 : -- link 2 in

  PROC monitor(CHAN OF ANY from.kb, to.monitor, to.screen,

  VAL stop.token          IS -1 (INT) :
  VAL shut.down.token     IS -2 (INT) :
  VAL time.token          IS -3 (INT) :
  VAL otherwise IS TRUE   :
  VAL node.no       IS     4     :
  VAL block.size    IS   100     :
  VAL max.iteration IS 1000      :
  VAL label IS "Virtual Shared Data Structure Test Harness"          :
  VAL shut.down IS "System Shut Down"        :
  VAL message.line IS 20    :
  INT node, system.count    :
  [block.size]INT  block    :
  [block.size * node.no]INT  array :
   [node.no]INT event.count :
   BOOL active              :
  TIMER clock               :
  INT   start. stop, start.point, granularity   :
  -- terminal driver constants
  VAL tt.go.to          IS  5 (BYTE) :
  VAL tt.out.string     IS  8 (BYTE) :
  VAL tt.beep           IS 13 (BYTE) :
  VAL tt.terminate      IS 15 (BYTE) :
  VAL tt.initialise     IS 17 (BYTE) :
  VAL tt.out.byte       IS 18 (BYTE) :
  VAL tt.out.int        IS 19 (BYTE) :
```

```
--------------------------------------------------------------------------------
PROC write.int(CHAN OF ANY to.screen,
                          VAL INT number, field)
  -- algorithm from Gerraint Jones
--------------------------------------------------------------------------------
  VAL tt.out.byte IS 18 (BYTE) :
  INT value, spaces, width :
  SEQ
  IF
    number >= 0
      SEQ
        spaces := -1
        width  := 1
    number < 0
      SEQ
        spaces := 1
        width  := 2
  WHILE (number / spaces) <= (-10)          -- calculate the width
    SEQ
      spaces := spaces * 10
      width  := width  + 1
  WHILE width < field             -- pad spaces
    SEQ
      to.screen ! tt.out.byte; ' '
      width  := width + 1
    IF                            -- place a minus sign if negative
      number < 0
        SEQ
          to.screen ! tt.out.byte; '-'
      otherwise
        SKIP
    WHILE spaces <> 0                 -- display numbers
      SEQ
        to.screen ! tt.out.byte; BYTE((INT '0') -
        spaces := spaces / 10
:
```

AD-A188 995 DESIGN IMPLEMENTATION AND EVALUATION OF A VIRTUAL SHARED MEMORY SYSTEM IN A MULTI-TRANSPUTER NETWORK(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA S J HART DEC 87 2/2

UNCLASSIFIED F/G 12/5 NL


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0

```
--------------------------------------------------
PROC clear.line(CHAN OF ANY to.screen)
--------------------------------------------------
  VAL tt.clear.eol IS 9 (BYTE) :
  SEQ
    to.screen ! tt.clear.eol
:
--------------------------------------------------
PROC go.to( CHAN OF ANY to.screen, VAL INT X,Y)
--------------------------------------------------
  VAL tt.go.to IS BYTE 5 :
  SEQ
    to.screen ! tt.go.to; X; Y
:
--------------------------------------------------
PROC write.s( CHAN OF ANY to.screen, VAL []BYTE s)
--------------------------------------------------
  VAL tt.out.string IS 8 (BYTE):
  SEQ
    to.screen ! tt.out.string; SIZE s
    to.screen ! s
:
--------------------------------------
PROC advance(INT event.counter)
--------------------------------------
  SEQ
    event.counter := event.counter + 1
:
```

```
PROC collect.data(CHAN OF ANY to.monitor,
                  to.data.structure)

  [node.no+1]CHAN OF ANY feed.pipe :
  PAR
    INT node, event.count, elapsed.time  :
    [block.size]INT sector       :
    SEQ
      to.monitor ? node
      IF
        node < 0
          SEQ
            IF
              node = time.token
                SEQ
                  to.monitor    ? elapsed.time
                  feed.pipe[0]   ! node; elapsed.time
              otherwise
                feed.pipe[0] ! node
        otherwise
          SEQ
            to.monitor   ? sector; event.count
            feed.pipe[0] ! node; sector; event.count
  PAR i = 0 FOR node.no
    INT node, event.count, elapsed.time        :
    [block.size]INT sector       :
    SEQ
      feed.pipe[i] ? node
      IF
        node <  0
          IF
            node = time.token
              SEQ
                feed.pipe[i]   ? elapsed.time
                feed.pipe[i+1] ! node; elapsed.time
            otherwise
              feed.pipe[i+1] ! node
        otherwise -- node >= 0
          SEQ
            feed.pipe[i]   ? sector ; event.count
            feed.pipe[i+1] ! node ; sector; event.count


      INT node, event.count, elapsed.time :
```

```
[block.size]INT sector      :
SEQ
  feed.pipe[node.no] ? node
IF
  node < 0
    IF
      node = time.token
        SEQ
          feed.pipe[node.no] ? elapsed.time
          to.data.structure !  node; elapsed.time
      otherwise
        to.data.structure ! node
  otherwise  -- node < 0
    SEQ
      feed.pipe[node.no] ? sector ; event.count
      to.data.structure ! node ; sector ; event.count
```

```
PROC update.memory( CHAN OF ANY in, to.screen, []INT array,
                    [node.no]INT event.count )
-- place sector in the virtual array according to the node
-- whence it came
--------------------------------------------------------------------------------
  VAL INT initial.value IS 0    :
  INT start.point, count, ticks :
  [block.size]INT  sector       : -- node responsibility
  INT node, elapsed.time        : -- which machine
  SEQ
    SEQ i= 0 FOR block.size
      sector[i] := -1
    active := TRUE
    in ? node
    IF
      node < 0
        IF
          node = shut.down.token
            SEQ
              go.to(to.screen, 28, 20)
              write.s(to.screen, "Shut.Down.Token Received")
              active := FALSE  -- process stops
          node = stop.token
            SEQ
              go.to(to.screen, 30, 22)
              write.s( to.screen, "Stop.Token Dispatched")
              clock ? stop  -- stop the system timer
          node = time.token
            SEQ
              in ? elapsed.time
              go.to(to.screen, 28, 3)
              write.int(to.screen, elapsed.time, 8)
          otherwise
            SKIP
      otherwise
        SEQ
          in ? sector ; event.count[node]
          start.point := node * block.size
          [array FROM start.point FOR block.size ] := sector
:

SEQ
  -- initialize data
  active     := TRUE
```

```
    system.count := 0
    SEQ i = 0 FOR node.no*block.size
      array[i] := -10
    SEQ i = 0 FOR node.no
      event.count[i] := 0

    -- handshake with network
    from.monitor ! max.iterations  -- start the process in the network
    to.monitor   ? granularity     -- get the networks granularity
    clock ? start   -- start the system timer

    go.to(screen, 40 - ((SIZE label)/2))  , 1 )
    write.s(to.screen, label)
    go.to(to.screen, 1, 4)
    write.s(to.screen, "Iterations #  ==> ")
    write.int(to.screen, iteration, 8)
    go.to(screen, 1, 5)
    write.s(to.screen, "Granularity ==> ")
    write.int(to.screen, granularity, 8)
    WHILE active
    SEQ
      CHAN OF ANY to.data.structure :
      PRI PAR
        collect.data00 (to.monitor, to.data.structure)
        update.memory(to.data.structure, to.screen, array,
      go.to(to.screen, 1 ,message.line)
      clear.line(to.screen)
      go.to(to.screen, (40 - ((SIZE shut.down)/2)), message.line)
      write.s(to.screen,  shut.down )
      -- display system elapsed time
      write.s(to.screen, "*C*N Elapsed Time is ==> ")
      write.int(to.screen, stop - start, 8)
:

SEQ
  monitor(keyboard, from.B003, screen, to.B003)
  INT ch :
  keyboard ? ch
:
```

LIST OF REFERENCES

[At87] Atkin, P. *Performance Maximization.* INMOS Technical Note Number 17, March 1987, Bristol, United Kingdom.

[Be85] Bekir, Evin. *Implementation of a Serial Delay Insertion Loop Communication for a Real Time Multitransputer System.* M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[Br87] Bryant, G. R. PASCAL *Source Code for Transputer Instruction Set Disassembler.*

[Ga86] Garret, D. R. *A Software System Implememtation Guide and System Prototyping Facility for the MCORTEX Executive on the Real Time Cluster,* M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1986.

[GiMi87] Gimarc, C. E., and Milutinovic, V. M. "A Survey of RISC Processors and Computers of the Mid 1980's." *IEEE Computer,* vol. 20, no. 9, pp. 59-69, September 1987.

[Hi] Hill, G. *Transputer Networks using the IMS B003.* INMOS Technical Note Number 13, undated, Bristol, United Kingdom.

[Ho79] Hoare, C. A. R. "Comunicating Sequential Processes." *Communications of the ACM,* vol. 21, no. 8, pp. 666-677, August 1978.

[In87a] *Transputer Reference Manual.* INMOS Ltd., January 1987, Bristol, United Kingdom.

[In87b] *T414 Preliminary Data Sheet.* INMOS Ltd., 1987, Bristol, United Kingdom.

[In87c] *T800 Preliminary Data Sheet.* INMOS Ltd., 1987, Bristol, United Kingdom.

[In87d] *The Transputer Instruction Set—A Compiler Writers Guide.* INMOS Ltd., February 1987, Bristol, United Kingdom.

[In87e] *INMOS Databook '87.* INMOS Ltd., Bristol, United Kingdom.

[Ko83] Kodres, U. R. "Processing Efficiency of a Class of Multi-computer Systems," *International Journal of Mini and Micro-computers*, vol. 5, no. 2, pp. 28-33, 1983.

[Ma87] Discussion with Philip Mattos, Applications Engineer, INMOS Ltd., October 1987.

[MaSh87a] May, D., and Shepherd, R. *Communicating Process Computers.* INMOS Technical Note Number 2, February 1987, Bristol, United Kingdom.

[MaSh87b] May, D., and Shepherd, R., *The Transputer Implementation of OCCAM.* INMOS Technical Note Number 21, February 1987, Bristol, United Kingdom.

[Pa] Packer, J. *Exploiting Concurrency: A Ray Tracing Example.* INMOS Technical Note Number 7, undated, Bristol, United Kindom.

[Po85] Pountain R. "Turbocharging MandelBrot." *BYTE Magazine*, vol. 10, no. 9, pp. 359-366, September 1986.

[Po87] Poole, M. *OCCAM Program Development Using the IMS D701 Transputer Development System.* INMOS Technical Note Number 16, January 1987, Bristol, United Kingdom.

[PoMa87] Pountain, R., and May, D. *A Tutorial Introduction to OCCAM Including Language Definition.* INMOS Ltd., March 1987, Bristol, United Kingdom.

[ReKa79] Reed, D. P., and Kanodia, R. K. "Synchronization with Eventcounts and Sequencers." *Communications of the ACM*, vol. 22, no. 2, pp. 115-123, February 1979.

[ShWA87] Shatz, S. M., and Wang, J. P. "Introduction to Software Engineering." *IEEE Computer*, vol. 10, no. 9, pp. 23-31, October 1987.

[Va87] Vanni, Filho J. *Test and Evaluation of the Transputer in a Multitransputer Configuration.* M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1987.

[We80] Weitzman, C. *Distributed Micro/Minicomuter Systems.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1980.

[Wi87a] Wilson, P. *An Introduction to Transputers.* INMOS Ltd., draft, 1.0, Colorado Springs, Colorado, USA, September 1987.

[Wi87b] Discussion with Peter Wilson, Strategic Applications Engineer, INMOS Ltd., Colorado Springs, Colorado, USA, November 1987.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
4.	Dr. Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, CA 93943	3
5.	Major Richard A. Adams, USAF, Code 52Ad Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
6.	Daniel Green, Code 20F Naval Surface Weapons Center Dahlgren, VA 22449	1
7.	Jerry Gaston, Code N24 Naval Surface Weapons Center Dahlgren, VA 22449	1
8.	Captain J. Hood, USN PMS 400B5 Naval Sea Systems Command Washington, DC 20362	1

9. RCA AEGIS Repository — 1
RCA Corporation
Government Systems Division
Mail Stop 127-327
Moorestown, NJ 08057

10. Library (Code E33-05) — 1
Naval Surface Weapons Center
Dahlgren, VA 22449

11. Dr. M. J. Gralia — 1
Applied Physics Laboratory
Johns Hopkins Road
Laurel, MD 20702

12. Dana Small, Code 8242 — 1
Naval Ocean Systems Center
San Diego, CA 92152

13. Director Naval Weapons Design — 1
Department Of Defence (Navy)
CP 1-6-18
Campbell Park Offices
Russell, ACT 2600
AUSTRALIA

14. Australian Defence Force Academy — 1
Duntroon, ACT 2600
AUSTRALIA

15. Dr. W.G.P. Robertson — 1
Director, WSRL
GPO BOX 2151
Adelaide, South Australia 5001
AUSTRALIA

16. Mr Neil Mitchel — 1
INMOS Corporation
2620 Augustine Drive, Suite 180
Santa Clara, CA 95054

17. Steve Burns — 1
INMOS CORPORATION
P.O. Box 16000
Colarado Springs, CO 80935-1600

18. Lieutenant Commander J. Vanni Filho, 1
Brazilian Navy
c/o Brazilian Naval Commision (DACM)
4706 Wisconsin Avenue, N.W.
Washington, DC 20016

19. Lieutenant Commander S. J. Hart, 3
Royal Australian Navy
Combat Data Systems Centre
84 Maryborough Street
Fyshwick, ACT 2610
AUSTRALIA

20. Dr. R. J. Dyne 1
Attaché (Defence Science)
Embassy of Australia
1601 Massachusets Avenue
Washington DC 20016

21. Mr. E. Carrapezza, Code 52 1
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

22. AEGIS Modelling Laboratory, Code 52 3
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

END

FILMED

MARCH, 1988

DTIC